



UNIVERSITÄT
DES
SAARLANDES

Evidence-driven Testing and Debugging of Software Systems

by

Ezekiel Olamide Soremekun

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

Saarbrücken

2021

Evidence-driven Testing and Debugging of Software Systems
Dissertation from Ezekiel Olamide Soremekun
Saarbrücken
April 2021

Day of Colloquium:	April 8th, 2021
Dean of the Faculty	Univ.-Prof. Dr. Thomas Schuster
Chair of the Committee:	Prof. Dr. Christian Rossow
Reporters	
First reviewer:	Prof. Dr. Andreas Zeller
Second reviewer:	Dr. Marcel Böhme
Third reviewer:	Prof. Dr. Lars Grunske
Academic Assistant:	Dr. Rafael Dutra

Abstract

Program debugging is the process of testing, exposing, reproducing, diagnosing and fixing software bugs. Many techniques have been proposed to aid developers during software testing and debugging. However, researchers have found that developers hardly use or adopt the proposed techniques in software practice. Evidently, this is because there is a *gap between proposed methods and the state of software practice*. Most methods fail to address the actual needs of software developers. In this dissertation, we pose the following scientific question: *How can we bridge the gap between software practice and the state-of-the-art automated testing and debugging techniques?*

To address this challenge, we put forward the following thesis: *Software testing and debugging should be driven by empirical evidence collected from software practice*. In particular, we posit that the *feedback from software practice should shape and guide (the automation) of testing and debugging activities*. In this thesis, we focus on gathering evidence from software practice by conducting several empirical studies on software testing and debugging activities in the real-world. We then build tools and methods that are well-grounded and driven by the empirical evidence obtained from these experiments.

Firstly, we conduct an empirical study on the *state of debugging in practice* using a survey and a human study. In this study, we ask developers about their debugging needs and observe the tools and strategies employed by developers while testing, diagnosing and repairing real bugs. Secondly, we evaluate the *effectiveness of the state-of-the-art automated fault localization (AFL)* methods on real bugs and programs. Thirdly, we conducted an experiment to evaluate the causes of *invalid inputs in software practice*. Lastly, we study how to learn input distributions from real-world sample inputs, using probabilistic grammars.

To bridge the gap between software practice and the state of the art in software testing and debugging, we proffer the following empirical results and techniques: (1) We collect evidence on the *state of practice in program debugging* and indeed, we found that there is a chasm between (available) debugging tools and developer needs. We elicit the actual needs and concerns of developers when testing and diagnosing real faults and provide a benchmark (called DBGBENCH) to aid the automated evaluation of debugging and repair tools. (2) We provide empirical evidence on the *effectiveness of several state-of-the-art AFL techniques* (such as statistical debugging formulas and dynamic slicing). Building on the obtained empirical evidence, we provide a hybrid approach that outperforms the state-of-the-art AFL techniques. (3) We evaluate the prevalence and causes of *invalid inputs in software practice*, and we build on the lessons learned from this

experiment to build a general-purpose algorithm (called *ddmax*) that automatically diagnoses and repairs real-world invalid inputs. (4) We provide a method to learn the *distribution of input elements in software practice* using probabilistic grammars and we further employ the learned distribution to drive the test generation of inputs that are similar (or dissimilar) to sample inputs found in the wild.

In summary, we propose an evidence-driven approach to software testing and debugging, which is based on collecting empirical evidence from software practice to guide and direct software testing and debugging. In our evaluation, we found that our approach is effective in improving the effectiveness of several debugging activities in practice. In particular, using our evidence-driven approach, we elicit the actual debugging needs of developers, improve the effectiveness of several automated fault localization techniques, effectively debug and repair invalid inputs, and generate test inputs that are (dis)similar to real-world inputs. Our proposed methods are built on empirical evidence and they improve over the state-of-the-art techniques in testing and debugging.

Keywords : *Software Testing, Automated Debugging, Automated Fault Localization, Input Debugging, Grammar-based Test generation*

Zusammenfassung

Software-Debugging bezeichnet das Testen, Aufspüren, Reproduzieren, Diagnostizieren und das Beheben von Fehlern in Programmen. Es wurden bereits viele Debugging-Techniken vorgestellt, die Softwareentwicklern beim Testen und Debuggen unterstützen. Dennoch hat sich in der Forschung gezeigt, dass Entwickler diese Techniken in der Praxis kaum anwenden oder adaptieren. Das könnte daran liegen, dass es einen großen Abstand zwischen den vorgestellten und in der Praxis tatsächlich genutzten Techniken gibt. Die meisten Techniken genügen den Anforderungen der Entwickler nicht. In dieser Dissertation stellen wir die folgende wissenschaftliche Frage: Wie können wir die Kluft zwischen Software-Praxis und den aktuellen wissenschaftlichen Techniken für automatisiertes Testen und Debugging schließen?

Um diese Herausforderung anzugehen, stellen wir die folgende These auf: Das Testen und Debuggen von Software sollte von empirischen Daten, die in der Software-Praxis gesammelt wurden, vorangetrieben werden. Genauer gesagt postulieren wir, dass das Feedback aus der Software-Praxis die Automation des Testens und Debuggens formen und bestimmen sollte. In dieser Arbeit fokussieren wir uns auf das Sammeln von Daten aus der Software-Praxis, indem wir einige empirische Studien über das Testen und Debuggen von Software in der echten Welt durchführen. Auf Basis der gesammelten Daten entwickeln wir dann Werkzeuge, die sich auf die Daten der durchgeführten Experimente stützen.

Als erstes führen wir eine empirische Studie über den Stand des Debuggens in der Praxis durch, wobei wir eine Umfrage und eine Humanstudie nutzen. In dieser Studie befragen wir Entwickler zu ihren Bedürfnissen, die sie beim Debuggen haben und beobachten die Werkzeuge und Strategien, die sie beim Diagnostizieren, Testen und Aufspüren echter Fehler einsetzen. Als nächstes bewerten wir die Effektivität der aktuellen Automated Fault Localization (AFL)-Methoden zum automatischen Aufspüren von echten Fehlern in echten Programmen. Unser dritter Schritt ist ein Experiment, um die Ursachen von defekten Eingaben in der Software-Praxis zu ermitteln. Zuletzt erforschen wir, wie Häufigkeitsverteilungen von Teileingaben mithilfe einer Grammatik von echten Beispiel-Eingaben aus der Praxis gelernt werden können.

Um die Lücke zwischen Software-Praxis und der aktuellen Forschung über Testen und Debuggen von Software zu schließen, bieten wir die folgenden empirischen Ergebnisse und Techniken: (1) Wir sammeln aktuelle Forschungsergebnisse zum Stand des Software-Debuggens und finden in der Tat eine Diskrepanz zwischen (vorhandenen) Debugging-Werkzeugen und dem, was der Entwickler tatsächlich benötigt. Wir sammeln die tatsächlichen Bedürfnisse von Entwicklern beim Testen und

Debuggen von Fehlern aus der echten Welt und entwickeln einen Benchmark (DBGBENCH), um das automatische Evaluieren von Debugging-Werkzeugen zu erleichtern. (2) Wir stellen empirische Daten zur Effektivität einiger aktueller AFL-Techniken vor (z.B. Statistical Debugging-Formeln und Dynamic Slicing). Auf diese Daten aufbauend, stellen wir einen hybriden Algorithmus vor, der die Leistung der aktuellen AFL-Techniken übertrifft. (3) Wir evaluieren die Häufigkeit und Ursachen von ungültigen Eingaben in der Softwarepraxis und stellen einen auf diesen Daten aufbauenden universell einsetzbaren Algorithmus (*ddmax*) vor, der automatisch defekte Eingaben diagnostiziert und behebt. (4) Wir stellen eine Methode vor, die Verteilung von Schnipseln von Eingaben in der Software-Praxis zu lernen, indem wir Grammatiken mit Wahrscheinlichkeiten nutzen. Die gelernten Verteilungen benutzen wir dann, um den Beispiel-Eingaben ähnliche (oder verschiedene) Eingaben zu erzeugen.

Zusammenfassend stellen wir einen auf der Praxis beruhenden Ansatz zum Testen und Debuggen von Software vor, welcher auf empirischen Daten aus der Software-Praxis basiert, um das Testen und Debuggen zu unterstützen. In unserer Evaluierung haben wir festgestellt, dass unser Ansatz effektiv viele Debugging-Disziplinen in der Praxis verbessert. Genauer gesagt finden wir mit unserem Ansatz die genauen Bedürfnisse von Entwicklern, verbessern die Effektivität vieler AFL-Techniken, debuggen und beheben effektiv fehlerhafte Eingaben und generieren Test-Eingaben, die (un)ähnlich zu Eingaben aus der echten Welt sind. Unsere vorgestellten Methoden basieren auf empirischen Daten und verbessern die aktuellen Techniken des Testens und Debuggens.

Schlüsselwörter : *Testen, Automatisches Debuggen, Automatisches Finden von Fehlern, Eingabeberichtigung, Grammatikbasiertes Testen*

Dedication

To God and my loving parents.

Acknowledgment

Firstly, I want to appreciate my advisor, Prof. Dr. Andreas Zeller. You have been an excellent mentor; your research ideas, guidance, communication skills and passion for good research have been a major motivation to me. It has been a great opportunity to work with you in the last few years. More importantly, I am thankful that beyond work, you paid attention to my welfare.

I am grateful to the members of the thesis committee (Prof. Dr. Lars Grunske and Dr. Marcel Böhme) for accepting to review this thesis. I appreciate the contribution of close collaborators (Lukas Kirschner, Marcel, Lars, Dr. Esteban Pavese, Nikolas Havrikov, Emamurho Ugherughe, Dr. Sudipta Chattopadhyay), thank you for your contributions to this body of work, I am grateful for the interesting research endeavors. To my office mates over the years – Björn Mathis, Alexander Kampmann, Nikolas Havrikov, Matthias Höschele and Clemens Hammacher, I am thankful for the daily conversations and your help with ideas and technical issues. Notably, I want to appreciate Björn and Michaël Mera, for constructive feedback on research projects and papers. I am also thankful for the Dagstuhl trips, and lunch break discussions with my colleagues (Nataniel Borges, Konstantin Kuznetsov, Andreas Rau, Jenny Rau and Sascha Just) and Post Docs (Alessio Gambi, Rahul Gopinath and María Gómez Lacruz) in the software engineering group. I want to also appreciate my first research mentors and advisers (Sudipta and Marcel), I learned a lot working and interacting with both of you. I am particularly grateful to Marcel for introducing me to software engineering (SE) research by advising my research immersion on automated debugging. I am also grateful for the research internship with Sudipta, it was an interesting introduction to research in the intersection of machine learning (ML) and SE. I want to appreciate Sakshi Udeshi for making my Singapore research trip fun and for the interesting weekly research discussions.

To my closest friends in Nigeria – Isa, Kunle, Biodun and Teye. Thank you for always making my Nigerian holidays a good break from the continuous grind of research. To my Nigerian friends in Germany (the Ratata family, especially Onyi, Bright, Emamurho, Eustace, David, Olachi, Tejumade, Chinyere, Anita, Bobo, Mark, Yakubu and Fuji), thank you for the frequent Jollof rice and game nights. To Ufuoma Bright Ighoroje, for encouraging me to apply for this PhD program, and ensuring I applied on the very last day of the deadline. Thank you for always being a mentor in many ways than one. To Bobo and Mark, thank you for the (online) FIFA game nights. To my closest friends from the graduate school (Marko, Apratim, Kireeti, Shweta and Dajana), I am grateful I met you all, you made my weekends enjoyable. I would not forget all the European trips, bar visits and nerdy arguments. To my best friend (Onyi) and her husband (Bryan), thank

you for your support since my first day in Saarbrücken.

I am thankful to God for my family, my partner (Matilda), my sisters (Grace and Helen) and my parents (Samuel and Christianah). I can not over-emphasize the importance of the emotional support and understanding of my sisters, thank you for bringing sunshine to my life, especially on my very few Christmas trips. To my partner – Matilda, for always being there for me, for the love and support. Thank you for keeping me focused on writing this thesis and tracking my progress, since I am always distracted by the next research project. This process would have been much harder without you.

I dedicate this thesis to God and my parents. To God, without whom I will not have made it this far, I am grateful for the favor, grace and endless mercies. To my parents, Samuel and Christianah, for their love, trust and patience, as well as your time and financial investments in my education and upbringing. Daddy (Samuel), thank you for being a good example. You taught me to be focused, generous, happy and humble. You have taught me to always focus first on the most important things in life, and to always invest my energy and time (in your words) in “certainties” rather than “uncertainties”. Thank you for teaching me the discipline to not be driven by money, time and location. Mom (Christianah), you have taught me patience, calmness and perseverance. I appreciate your love and the privileges you afford me, you are a source of joy. This PhD program would have been significantly difficult without your support and trust.

Contents

1	Introduction	1
1.1	Thesis Statement	2
1.2	Publications	3
1.3	Dissertation Outline	4
2	Background	5
2.1	Software Failures	5
2.2	Problem Statement	6
2.3	Debugging Process	7
2.4	Debugging Challenges	8
2.4.1	Debugging in Software Practice	8
2.4.2	Fault Localization	8
2.4.3	Input Debugging	13
2.4.4	Test Generation	14
3	Debugging in Practice: An Empirical Study	17
3.1	Introduction	17
3.2	Survey	19
3.2.1	Study Design	19
3.2.2	Study Results	23
3.3	Observational Study	33
3.3.1	Study Design	34
3.3.2	Study Results	40
3.4	A Benchmark for Debugging Tools	48
3.5	Limitations and Threats to Validity	52
3.6	Related Work	54
3.7	Discussions and Future Work	56
4	Locating Faults with Program Slicing: An Empirical Analysis	61
4.1	Introduction	61
4.2	A Hybrid Approach	65

4.3	Evaluation Setup	67
4.3.1	Implementation	68
4.3.2	Metrics and Measures	69
4.3.3	Objects of Empirical Analysis	71
4.3.4	Measure of Localization Effectiveness	73
4.4	Experimental Results	76
4.5	Threats to Validity	91
4.6	Related Work	92
4.7	Discussions and Future Work	94
5	Debugging Failure-Inducing Inputs	97
5.1	Introduction	97
5.2	Prevalence of Invalid Inputs	99
5.3	Lexical Repair	102
5.4	Syntactic Repair	110
5.5	Diagnostic Quality	114
5.6	Threats to Validity	116
5.7	Limitations	116
5.8	Related Work	117
5.9	Discussions and Future Work	118
6	Learning Input Distributions for Grammar-Based Test Generation	121
6.1	Introduction	121
6.2	Overview	123
6.3	Approach	126
6.3.1	Probabilistic Grammars	126
6.3.2	Learning Probabilities	127
6.3.3	Inverting Probabilities	127
6.3.4	Producing Inputs from a Grammar	128
6.3.5	Implementation	129
6.4	Experimental Evaluation	130
6.4.1	Evaluation Setup	131
6.4.2	Experimental Results	135
6.5	Threats to Validity	145
6.6	Limitations	146
6.7	Related Work	146
6.8	Discussions and Future Work	148
7	Conclusion	151
7.1	Summary	151
7.2	Contributions	152
7.3	Future Work	153

Bibliography	157
Appendices	181
List of Figures	183
List of Tables	187

Introduction

“Beware of bugs in the above code;
I have only proved it correct, not tried it.”
— Donald E. Knuth

Programs often fail. Frequently, programs contain bugs that cause failures and unexpected behaviors. When a program fails, software developers are saddled with the task of exposing, reproducing, diagnosing and repairing the bug. Developers first have to test the program with suitable inputs, then diagnose and fix the failure. *Testing* is the process of writing or generating inputs that executes the program code, in order to expose (or reproduce) bugs. Meanwhile, *debugging* is the process of diagnosing and fixing such bugs. Both testing and debugging are arduous tasks, that consume a lot of time and resources.

Several methods have been developed to support developers in software testing and debugging [1, 2, 3, 4, 5, 6, 7, 8]. However, *there is a gap between proposed methods and the actual state of software practice*. Most methods lack strong empirical support; they either do not have any empirical validation at all or have weak evaluations with unrealistic assumptions [9, 10]. Often, researchers have found evidence of assumptions, tools and techniques that do not apply in practice [11, 12, 13, 14, 15]. This lack of empirical support can be attributed to the cost and difficulty of experimentation [16, 17, 18].

Consequently, this chasm has created *a gap between the state of the art tools and how developers actually test and debug programs*. For instance, a recent survey on debugging cites more than 400 publications on fault localization techniques [1]. However, most developers have never used an automated fault localization tool in practice [19]. Likewise, many test generation methods do not generate test inputs that are similar to those written by developers or end-users of the software [20]. Notably, current approaches ignore how proposed tools address real-world needs. We know very little about how developers test and debug; we also lack the data and methods to check (proposed) tools against practitioner’s needs.

How can we bridge this gap? In this thesis, we conduct several empirical studies to gather evidence from software practice, in order to effectively guide and automate software testing and debugging. We put forward the thesis that *testing and debugging should be driven by empirical evidence collected from software practice*. We posit that the feedback from software practice

should shape and guide software testing and debugging. Thus, we focus on building tools and methods that are driven by empirical evidence collected from software practice.

1.1 Thesis Statement

In this dissertation, we designed and conducted several experiments to empirically test our thesis statement. Then, we collect empirical evidence from these experiments to build novel methods for testing and debugging. We analyze the results of our experiments in order to support the thesis, we discuss the implications of these experiments and we introduce novel testing and debugging methods that are built on empirical evidence from software practice.

The rest of this dissertation builds on different aspects of this thesis statement:

*Software testing and debugging should be driven
by empirical evidence collected from software practice.*

Firstly, we evaluate *the state of debugging in practice* (Chapter 3). In particular, we evaluate how developers diagnose and repair bugs in practice. We conducted a retrospective study with hundreds of developers and an observational study with 12 practitioners, in order to *collect empirical evidence on the state-of-the-practice in debugging*. We investigate how developers spend their time diagnosing and fixing bugs. We also examine the tools and strategies employed by developers while debugging. Our findings reveal the need for automated assistance to evaluate and develop realistic debugging aids. Subsequently, we apply the evidence gathered from these empirical studies to facilitate and guide future research. In particular, we provide DBGBENCH, a highly usable debugging benchmark that provides fault locations, patches and explanations for common bugs as provided by practitioners.

Secondly, we evaluate the *effectiveness of the state-of-the-art fault localization methods* (Chapter 4). Specifically, we compare the effectiveness of several statistical fault localization methods against dynamic program slicing in a large study of over 706 faults in 46 open source C programs. Our findings reveal that dynamic slicing was more effective than the best performing statistical debugging formula. For most bugs, dynamic slicing will find the fault earlier than the best performing statistical debugging formula. Consequently, we apply this empirical evidence to develop a *hybrid approach*. The hybrid approach leverages the strengths of both dynamic slicing and statistical fault localization to achieve the best results. Programmers using the hybrid approach will need to examine fewer lines of code to locate faults.

Thirdly, we evaluate *how to debug invalid inputs that occur in software practice* (Chapter 5). We first evaluate the prevalence of invalid inputs in practice, and we found that four percent of inputs in the wild are invalid. We then build on this empirical evidence to develop a method to debug inputs — that is, identify which parts of the input data prevent processing, and recover as much of the (valuable) input data as possible. We present a general-purpose algorithm called *ddmax* that addresses these problems automatically, via numerous experiments that maximizes the subset of the input that can still be processed by the program. Particularly, this is the first approach that fixes faults in the input data without requiring program analysis.

Lastly, we propose an approach that *generates structured inputs that are (dis)similar to real-world inputs found in the wild* (Chapter 6). We first learn the distribution of input elements in

real-world sample inputs using a probabilistic grammar. We then apply the learned probabilistic grammar to generate inputs that are similar to the sample. We also generate inputs that are dissimilar to the sample by inverting the learned probabilities. In addition, this approach allows to reproduce (or avoid) program failures by learning the distribution of input elements in failure-inducing inputs.

1.2 Publications

This dissertation builds on the following papers (in chronological order):

- **Ezekiel Soremekun**. Debugging with probabilistic event structures. In 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pages 437–440. IEEE, 2017.
- Marcel Böhme, **Ezekiel Soremekun**, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Wo ist der fehler und wie wird er behoben? ein experiment mit softwareentwicklern. Software Engineering und Software Management (SESWM), 2018.
- Marcel Böhme, **Ezekiel Soremekun**, Sudipta Chattopadhyay, Emamurho Juliet Ugherughe, and Andreas Zeller. How developers debug software —the dbgbench dataset. In 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pages 244–246. IEEE, 2017.
- Marcel Böhme, **Ezekiel Soremekun**, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Where is the bug and how is it fixed? an experiment with practitioners. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pages 117–128, 2017.
- **Ezekiel Soremekun**, Lukas Kirschner, Marcel Böhme, and Andreas Zeller. Locating faults with program slicing: An empirical analysis. Empirical Software Engineering (EMSE) 26, 51 (2021).
- Lukas Kirschner, **Ezekiel Soremekun**, and Andreas Zeller. Debugging inputs. In 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). IEEE, 2020.
- **Ezekiel Soremekun**, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. Inputs from hell: Learning input distributions for grammar-based test generation. Transaction on Software Engineering (TSE), 2020.

The author of this dissertation has also co-authored the following papers, these papers do not contribute to this dissertation:

- Rahul Gopinath, Alexander Kampmann, Nicolas Havrikov, **Ezekiel Soremekun** and Andreas Zeller. Abstracting Failure-Inducing Inputs. In 2020 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2020. ACM Distinguished Award.

- Alexander Kampmann, Nicolas Havrikov, **Ezekiel Soremekun** and Andreas Zeller. When does my Program do this? Learning Circumstances of Software Behavior. In Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2020.
- Björn Mathis, Vitalii Avdiienko, **Ezekiel Soremekun**, Marcel Böhme, and Andreas Zeller. Detecting information flow by mutating input data. In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 263-273. IEEE, 2017.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows: First, in Chapter 2, we provide *background* on the main areas of software testing and debugging that are addressed by this dissertation. In particular, we discuss the state of the art in *program debugging*, *automated fault localization*, *input debugging* and *grammar-based test generation*. In Chapter 3, we present an *empirical study on debugging in practice* in which we investigate how developers diagnose and fix real bugs. We present key findings on the tools and strategies used by developers in practice. We also provide DBGBENCH—a benchmark providing empirical data on debugging in practice, in order to aid future research. Chapter 4 introduces an empirical study of the effectiveness of the state-of-the-art *automated fault localization* techniques, in particular, program slicing and several statistical fault localization methods. We also present a hybrid approach that outperforms the state-of-the-art. Chapter 5 presents an empirical study of *invalid inputs* in the wild, where we present key insights on the prevalence and sources of input invalidity in software practice. Building on this evidence, we present a black-box approach that *repairs and debugs invalid inputs* using a maximizing variant of the delta debugging algorithm. In Chapter 6, we present a *grammar-based probabilistic test generation approach*, that allows to generate inputs that are (dis)similar to sample inputs written by end-users or developers. We further illustrate how our approach allows to generate inputs that reproduce (or avoid) failures. Finally, we conclude this dissertation with a discussion of the contributions and future directions in Chapter 7.

Background

This chapter is taken, directly or with minor modifications, from our 2017 ICSE paper *Debugging with probabilistic event structures* [21] and our 2018 SE paper *Wo ist der Fehler und wie wird er behoben? Ein Experiment mit Softwareentwicklern* [22]. My contribution in this work is as follows: (I) original idea; (II) partial implementation; (III) evaluation.

“If I have seen further, it is by standing on the shoulders of giants.”

— Sir Isaac Newton

2.1 Software Failures

When a program fails, we say the program is *buggy*. A program can be buggy by construction (e.g. due to omission errors) or because it contains defects. Invariably, for a software to be considered buggy, the execution of the software has to result in unexpected behavior(s).

Exposing bugs in programs is non-trivial and determining the presence of a bug is difficult. In software development, bugs are typically revealed via *software testing*. Developers have to construct test input(s) that expose and reproduce the observed unexpected behavior (e.g. a failure, crash or wrong output). A bug is found or exposed, when a program fails when fed with a *valid test input*. The failing test execution confirms that the program is *buggy*. Test inputs that cause a program to fail are referred to as *failure-inducing input*. These failure-inducing inputs allow developers to reproduce and diagnose the bug.

When a bug is found in a software, developers have to *debug the software*, i.e. understand, diagnose and fix the bug. The process of bug diagnosis and fixing is called *debugging*. Debugging is a tedious task that involves analyzing the *failing* program execution(s). When debugging, developers often employ techniques that aid program understanding, bug analysis and test execution.

Terminology

In the following, we explain the main debugging terminologies used in this dissertation based on the IEEE Glossary [23]. We define an *error* as incorrect program behavior or result, for instance, when the program prints 0 when it is expected to print 1. A *fault* is the defect in the

source code that causes an error, like a missing increment. A *bug* may be any of the above. An error’s *symptom* is the observed unexpected behavior, for instance, the program unexpectedly printing 0. A failing *test case* is an input and an assertion of the expected output that fails in the presence of an error. *Debugging* is a “post-mortem” activity that follows once an error has been found and reported, for instance, via *testing* or by a user. Debugging is the task of identifying the fault and removing the error completely without introducing new errors. We distinguish four distinct sub-tasks during debugging: *Bug reproduction* is the task of reproducing the bug locally by *constructing a test case* that fails because of the bug. *Bug diagnosis* is the task of *understanding and explaining the run-time actions causing the symptom*. *Fault localization* is the task of identifying the faulty code locations, i.e. the statements responsible for the bug and (potentially) have to be changed to fix it. Finally, *bug fixing or repair* is the task of removing the error by changing the source code.

2.2 Problem Statement

Several techniques have been proposed to support developers during debugging activities [1]. However, evidence has shown that most of these techniques are not widely adopted by developers in practice [11]. More importantly, the *state-of-the-art* approaches do not meet the real-world needs of developers [12]. In this dissertation, we ask the following fundamental question: *How can we bridge the gap between the state-of-the-art debugging methods and the state of debugging in practice?*

Indeed, addressing this question requires analyzing the state of debugging in practice and the performance of the state-of-the-art debuggers. On one hand, it is pertinent to conduct empirical studies that sheds light on the *nature of debugging and the needs of developers*. The empirical study should address questions such as: *How do developers debug programs?* And *what do developers need when debugging in practice?* The results of the study should provide the debugging requirements of developers and serve as a baseline to design and evaluate debugging aids. On the other hand, one needs to evaluate the *performance of the state-of-the-art debugging tools* in real world setting. In this context, one needs to address the following fundamental question: *What is the most effective fault localization method on real faults?* Subsequently, one can build methods that address the needs and concerns of developers elicited in the first step, in order to improve the performance of the state-of-the-art debugging tools in practical settings.

The main goal of this work is to provide methods that meet the actual needs of developers when debugging in practice. To achieve this goal, we employ an *evidence-based approach* to develop techniques which support developers during debugging activities. This is a two-step approach that proceeds for a debugging challenge or activity as follows: First, 1) conduct realistic empirical studies that *elicit the nature and requirements* of the debugging activity in practice, then 2) *develop methods and tools* that are based on the *evidence (i.e. requirements) gathered from the empirical studies* (in step one). In this dissertation, all empirical studies are based on the first step (1) of this approach, and all proposed methods in this dissertation are results of the second step (2) of the evidence-based approach.

2.3 Debugging Process

The main goal of *automated debugging* is to support developers during debugging activities. Debuggers are designed to improve developer's effectiveness during bug diagnosis and fixing. The usefulness and applicability of debuggers is vital, considering that about a third of development time is spent on debugging activities [1]. Thus, software developers can benefit significantly from techniques that improve productivity during debugging.

Typically, to effectively debug a program, a developer needs to 1) expose or reproduce the bug (*via testing*), 2) understand the unexpected program behavior (called *bug diagnosis*), 3) determine the fault locations responsible for the observed failing behavior (called *fault localization*) and (4.) repair or fix the bug (called *program repair*). In the following, we discuss these debugging steps:

1. **Bug Reproduction:** When debugging, developers must first *expose or reproduce the failure*, in order to observe the unexpected behavior. This may involve running existing tests or constructing new tests that reproduce the failure. These tests are expected to accurately and reliably reveal the unexpected software behavior, this is necessary to establish the presence of a bug in the software.
2. **Bug Diagnosis:** Secondly, the developer has to diagnose the bug or failure. The goal of this step is to comprehend the conditions under which the bug can be observed. This involves program comprehension (i.e. understanding the program behavior) and analyzing test executions, for instance, comparing failing and passing executions. Other bug diagnosis approaches include (dynamic) program analysis, i.e. analyzing program executions in order to diagnose the conditions (e.g. data or control flow paths) under which the failing behavior can (not) be observed.
3. **Fault Localization:** In this step, the developer analyzes the program to determine the faulty code locations. The fault locations are the program features (e.g. statements or lines of code) that explains the bug, such features may need to be modified to fix the bug. This step is often achieved by analyzing the code locations that are executed in the failing (and passing) execution(s), in order to determine the code point(s) where the executions diverged from the expected behavior. Fault Localization is a prerequisite step to fix a bug, thus, it is the first step of most APR tools. Examples of prominent AFL techniques include statistical debugging, program slicing and mutational AFL.
4. **Program Repair:** Finally, the developer (or an APR tool) fixes the bug by modifying the program. This process patches the faulty program (e.g. by removing the error), in order to ensure that the program behavior matches the expected behavior. However, to make sure the fix is *correct*, it is important to ensure that 1) the failing test(s) no longer leads to a failure in the fixed program (called *patch plausibility*), 2) the fix is complete, it actually fixes the bug and not a symptom of the bug (called *patch completeness*) and 3) the fix (or program change) does not introduce a new bug or a regression bug that cause other tests to fail, this step is called *regression testing*.

2.4 Debugging Challenges

Let us provide relevant background on the challenges in the major areas of software testing and debugging that are investigated in this thesis. In particular, we discuss the state of the art in *program debugging*, *automated fault localization*, *input debugging* and *test generation*.

2.4.1 Debugging in Software Practice

In 1997, Liebermann introduced the CACM special section on “*The Debugging Scandal and What to Do About It*” with the words “debugging is still as it was 30 years ago, largely a matter of trial and error” [24]. Researchers argue that debugging in practice is an art more than a science [25], nothing but trial-and-error [24]. Others have gone as far as modelling the way how programmers debug as a predator that is following scent to find prey [26]. Except, debugging does not need to be an unpremeditated activity. Today, automated debugging tools hold the promise of providing guidance during debugging, of an increase of productivity, and of a significant reduction of time and money spent on debugging; all by the push of a button.

The “debugging scandal” is definitely not due to the lack of tools produced in the outstanding research of automated software debugging. Researchers have developed tools that can identify potential fault locations [27, 28], potential fix locations [29, 30], failure-inducing modifications in updates [31], the chain of events leading up to the error [32], and even potential software patches [33, 34, 35, 36]. However, empirical evidence shows that *most developers have never used an automated debugging tool* [19].

Why would a rational developer forgo the promises of automation in the face of the tediousness of the manual debugging activity? Andrew Ko writes about his experience as a CTO of a company and his attempts to bring software engineering research into practice that “most of what I present just isn’t relevant, isn’t believed, or isn’t ready; honestly, many of our engineering problems simply aren’t the problems that software engineering researchers are investigating” [37]. Lionel Briand adds that “the research community had a rather superficial understanding of the problems facing practitioners while debugging” [38].

In this thesis, we asked ourselves: what is the state of debugging in practice (see Chapter 3)? Which problems *do* practitioners have? What do *they* want? What *do* practitioners find relevant? *Why* do they reject debugging automation? In addition, we collected empirical data from hundreds of hours of debugging sessions of professional developers: We investigated how developers debug and repair real software bugs, the tools and strategies applied, the time taken and the bug diagnoses and patches they provided.

2.4.2 Fault Localization

This section sheds light on the details of *automated fault localization* (AFL) techniques. AFL refers to methods that automatically identify the *root cause(s)* of a bug, i.e. the *faulty program locations* (e.g. statements or methods) that triggered an observed failing behavior. The aim of AFL approaches is to provide developers (and automated tools) with faulty code locations, in order to aid bug diagnoses and program repair. AFL is an important software engineering research area, which has provided a wealth of automated tools. Wong et al. [39] surveyed more

than 60 AFL tools and list 29 as publicly available (e.g., as Eclipse plugin [40]). A recent survey finds that the majority¹ of practitioners considers research on AFL as essential or worthwhile [41].

In this dissertation, we investigate the efficacy of AFL approaches (see Chapter 4). There are two main AFL approaches, namely *program slicing* and *statistical debugging* [39]. In the rest of this section, we provide background on *program slicing* and *statistical debugging*.

Program Slicing

More than three decades ago, Mark Weiser [42, 43] noticed that developers localize the root cause of a failure by following chains of statements starting from where the failure is observed. Starting from the symptomatic statement s where the error is observed, developers would identify those program locations that directly influence the variable values or execution of s . This traversal continues transitively, until the root cause of the failure (i.e., the *fault*) is found. This procedure allows developers to investigate those parts of the program involved in the infected information-flow in reversed order towards the location where the failure is first observed.

Static Slicing

Weiser developed *program slicing* as the first automated fault localization technique. A programmer marks the statement where the failure is observed (i.e., the failure’s symptom) as *slicing criterion* C . To determine the potential impact of one statement onto another, the program slicer first computes the Program Dependence Graph (PDG) for the buggy program. The *PDG* is a directed graph with nodes for each statement and an edge from a node s to a node s' if

1. statement s' is a conditional (e.g., an *if*-statement) and s is executed in a branch of s' (i.e., the values in s' *control* whether or not s is executed), or
2. statement s' defines a variable v that is used at s and s may be executed after s' without v being redefined at an intermediate location (i.e., the values in s' *directly influence the value* of the variables in s).

The first condition elicits *control dependence* while the second elicits *data dependence*. The PDG essentially captures the information-flow among all statements in the program. If there is no path from node n to node n' , then the values of the variables at n have definitely no impact on the execution of n' or its variable values.

The *static program slice* [43, 44] computed with respect to C consists of all statements that are reachable from C in the PDG. In other words, it contains all statements that potentially impact the execution and program states of the slicing criterion. Note that static slicing only removes those statements that are *definitely not* involved in observing the failure at C . The statements in the static slice may or may not be involved. Static program slices are often very large [45].

Dynamic, Relevant, and Execution Slicing

A *dynamic program slice* [47, 48] is computed for a specific failing input t and is thus much smaller than a static slice. It is able to capture all statements that are *definitely* involved in

¹Majority here means above the median (i.e., 3rd and 4th quartile).

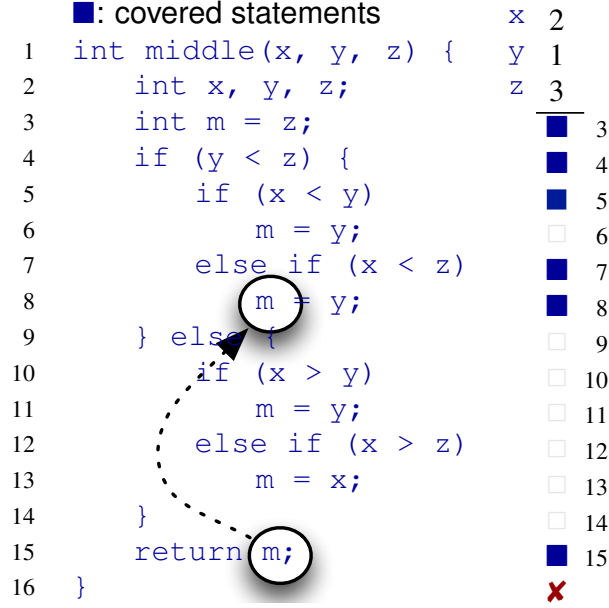


Figure 1: Dynamic slicing illustrated [46]: The `middle` return value in Line 15 is the slicing criterion and Line 8 is the faulty statement.

computing the values that are observed at the location where the failure is observed for failing input t . Specifically, the dynamic slice computed with respect to slicing criterion C for input t consists of all statements whose instances are reachable from C in the Dynamic Dependence Graph (DDG) for t . The *DDG* for t is computed similarly as the PDG, but the nodes are the statement *instances* in the execution trace $\pi(t)$. The DDG contains a separate node for each occurrence of a statement in $\pi(t)$ with outgoing dependence edges to only those statement instances on which this statement instance depends in $\pi(t)$ [48]. However, an error is not only explained by the actual information-flow towards C . It is important to also investigate statements that could have contributed towards an alternative, potentially correct information-flow. This is the main motivation for *relevant slicing*. The *relevant slice* [49, 50] computed for a failing input t subsumes the dynamic slice for t and also captures the fact that the fault may be in *not* executing an alternative, correct path. It adds conditional statements (e.g., `if`-statements) that were executed by t and if evaluated differently may have contributed to a different value for the variables at C . It requires computing (static) potential dependencies. In the execution trace $\pi(t)$, a statement instance s *potentially depends* on conditional statement instance b if there exists a variable v used in s such that (i) v is not defined between b and s in trace $\pi(t)$, (ii) there exists a path σ from $\varphi(s)$ to $\varphi(b)$ in the PDG along which v is defined, where $\varphi(b)$ is the node in the PDG corresponding to the instance b , and (iii) evaluating b differently may cause this untraversed path σ to be exercised. Qi et al. [51] proved that the relevant slice with respect to C for t contains *all* statements required to explain the value of C for t .

The *approximate dynamic slice* [52, 47] is computed with respect to slicing criterion C for failing input t as the set of *executed* statements in the static slice with respect to C . The approximate dynamic slice subsumes the dynamic slice because there can be an edge from an instance s to an instance s' in the DDG for t only if there is an edge from statement $\varphi(s)$ to statement $\varphi(s')$ in the PDG. The approximate dynamic slice subsumes the relevant slice because

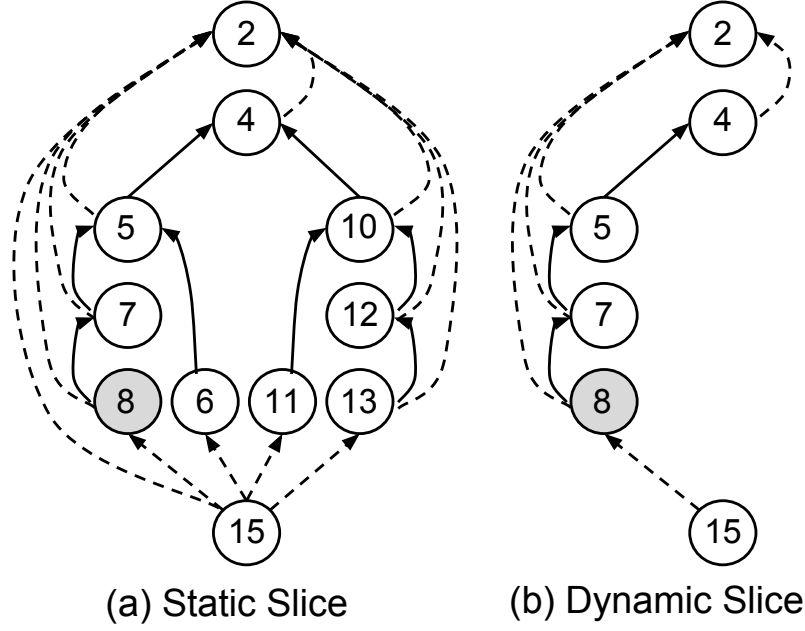


Figure 2: Slicing Example: Nodes are statements in each line of the `middle` program (Figure 1). Control-dependencies are dashed lines while data dependencies are shown as concrete lines.

it also accounts for potential dependencies: Suppose instance s potentially depends on instance b in execution trace $\pi(t)$. Then, by definition there exists a path σ from $\varphi(s)$ to $\varphi(b)$ in the PDG along at least one control and one data dependence edge (via the node defining v); and if $\varphi(s)$ is in the static slice, then $\varphi(b)$ is as well. Note that the approximate dynamic slice is (1) *easier to compute* than dynamic slices (static analysis), (2) *significantly smaller* than the static slice, and still (3) *as “complete”* as the relevant slice. In summary, $\text{dynamic slice} \subseteq \text{relevant slice} \subseteq \text{approximate dynamic slice} \subseteq \text{static slice}$.

Figure 2 (a) and (b) show the static and the dynamic slice for the `middle` program in Figure 1, respectively. The slicing criterion was chosen as the return statement of the program—that is the statement where the failure is observed. As test case, we chose the single failing test case $\mathbf{x} = 2$, $\mathbf{y} = 1$, and $\mathbf{z} = 3$. In this example, the approximate dynamic slice matches exactly the dynamic slice. In a debugging setting, programmers would follow *dynamic* dependencies to find those lines that actually impact the location of interest in the *failing run*. In our example (Figure 1), they could simply follow the dynamic dependency of Line 15 where the value of `m` is unexpected, and immediately reach the faulty assignment in Line 8—which again happens to be the faulty line. In this dissertation, we evaluated the fault localization effectiveness of dynamic slicing, we also compare its effectiveness to that of numerous statistical fault localization methods (see Chapter 4).

Statistical Debugging

Jones et al. introduced the first statistical debugging technique, TARANTULA [27], quickly followed by Liblit et al. [53, 54]. The main idea of *statistical debugging* is to associate the execution of a particular program element with the occurrence of failure using so-called *suspiciousness measures*. Program elements (like statements, basic blocks, functions, components, etc.) that are observed more often in failed executions than in correct executions are deemed as more suspicious. A

		Tarantula	Ochiai	Naish2
1	int middle(x, y, z) {			
2	int x, y, z;	0.500	0.408	0.167
3	int m = z;	0.500	0.408	0.167
4	if (y < z) {	0.500	0.408	0.167
5	if (x < y)	0.625	0.500	0.500
6	m = y;	0.000	0.000	-0.167
7	else if (x < z)	0.714	0.578	0.667
8	m = y;	0.833	0.707	0.833
9	} else {	0.000	0.000	-0.333
10	if (x > y)	0.000	0.000	-0.333
11	m = y;	0.000	0.000	-0.167
12	else if (x > z)	0.000	0.000	-0.167
13	m = x;	0.000	0.000	0.000
14	}	0.000	0.000	0.000
15	return m;	0.500	0.408	0.167
16	}			

Figure 3: Statistical Fault Localization Example: The faulty line 8 and its scores are in **bold red**

program element with a high suspiciousness score is more likely to be related to the root cause of the failure. When TARANTULA was first introduced, the authors envisioned an integrated development environment where more suspicious code regions are colored in different shades of red while less suspicious code regions are colored in shades of green. After the execution of the test suite, at first glance, the developer can identify likely code regions where the fault might hide.

An important property of statistical debugging is that apart from measuring coverage, it requires no specific static or dynamic program analysis. This makes it easy to implement and deploy, in particular as part of *automated program repair*. In fact, most popular automated repair techniques use statistical fault localization to automatically decide where to attempt the patch [33, 35, 55, 56]. Instead of a visualization that is presented to the developer, statistical fault localization provides a *ranking* that is presented to the automated repair technique. The highest ranked, most suspicious element is considered first as patch location. Using a more effective debugging technique thus directly increases the effectiveness of any repair technique.

Figure 3 shows the scores computed for the executable lines in our motivating example (Figure 1). The statement in Line 8 is incorrect and should read `m = x;` instead. This statement is also the most suspicious according to all three statistical fault localization techniques in the example. Notice that only twelve (12) lines are actually executable. Evidently, in this example from Jones and Harrold [46], the faulty statement is also the most suspicious for these three statistical fault localization techniques.²

In this dissertation, we investigate the effectiveness of the main families of statistical measures using 18 statistical fault localization formulas in total. There are four main families of statistical debugging formulas, namely *human-generated optimal* measures, *most popular* measures, *genetic programming* (GP) evolved measures and measures targeted at *single bug optimality*.

²The scores for the faulty statement in Line 8 are $\text{tarantula}(s_8) = \frac{1}{1} / (\frac{1}{1} + \frac{1}{5})$, $\text{ochiai}(s_8) = \frac{1}{\sqrt{1(1+1)}}$, and $\text{naish2}(s_8) = 1 - \frac{1}{1+4+1}$.

2.4.3 Input Debugging

Although there are hundreds of automatic fault localization (AFL) and repair techniques [1]; all of these techniques focus on *program code*, attempting to identify possible fault locations in the code and synthesizing fixes for this code. However, when a program fails on some input, it need not be the program code that is at fault. Often, the *input may be invalid*, e.g. due to hardware failures, hardware aging, transmission errors that corrupt input data. Input data can also be corrupted through software bugs, with the processing software writing out malformed or incomplete data. General-purpose automated debugging techniques (such as AFL) are not useful to debug inputs, since they focus on faults in code, they would regularly identify the input parser and its error-handling code as being associated with the fault.

In this dissertation, we investigate the problem of *input debugging*, i.e. *isolating faults in inputs* and *recovering as much data as possible* from the existing input (see Chapter 5). The aim is to *recover maximal valid data* from the failure-inducing input, as well as provide a *precise diagnosis or root cause of the input invalidity*. The rest of this section provides background on input debugging, we discuss the state-of-the-art techniques for identifying or fixing faults in inputs.

Input Minimization

We discuss the state of the art in *input reduction*. These techniques aim to produce a *subset* of the input that still produces the failure. These approaches isolate the faults in invalid or failure-inducing inputs, in order to obtain the minimal subset that reproduces a failure.

Researchers have proposed several methods to automatically identify failure causes in input [57, 58, 59, 60]. The *Delta Debugging* algorithm *ddmin* [57] reduces inputs by repeatedly removing parts of the input (first larger ones, then smaller ones) and testing whether running the program with the reduced input still produces the failure. The result is a so-called *1-minimal input* in which every single element (character) of the input is relevant for producing the failure. Some approaches employ the input structure to simplify inputs [59] (HDD) and [60]. In particular, [60] applies the syntactic structure of Java programs to simplify buggy Java programs [60]. Meanwhile, HDD [59] uses an hierarchical input minimization algorithm based on *ddmin*.

In this dissertation, we propose a maximizing variant of the *delta debugging* algorithm to provide precise diagnosis of failure-inducing inputs. Our algorithm works at both the lexical and syntactic levels of the input. In addition, we compare the performance of our proposed algorithm (*ddmax*) to that of *ddmin*.

Input Repair

The problem of *repairing* inputs is the *inverse* of *input minimization*. Rather than minimizing failure-inducing inputs (the reduction problem), users would be interested in *maximizing non-failure-inducing inputs*, i.e. *recover* as much data as possible from the input at hand.

There are a few approaches for repairing failure-inducing inputs. In security-critical applications, *input rectification* is employed to transform potentially malicious inputs, in order to ensure they behave safely. Input *rectifiers* [61, 62] address this problem by learning a set of constraints

from typical inputs, then transforming a malicious input into a benign input that satisfies the learned constraints. Other approaches apply program analysis to repair failure inducing inputs. In particular, Docoverly [63] and S-DAGS [64] are white-box techniques for fixing broken input documents. Docoverly [63] uses symbolic execution to manipulate corrupted input documents in a manner that forces the program to follow an alternative error-free path. S-DAGS [64] is a semi-automatic technique that enforces formal (semantic) consistency constraints on inputs documents in a collaborative document editing scenario. Likewise, *data diversity* [65] applies program analysis to transform an invalid input into a valid input that generates an equivalent result, in order to improve *software reliability*. This is achieved by finding the regions of the input space that causes a fault, and re-expressing a failing input to avoid the faulty input regions.

Unlike the aforementioned approaches, in this dissertation, we propose a black-box generic input repair approach that recovers the maximal passing input data, and provides the minimal input diagnoses using several test experiments (see Chapter 5).

2.4.4 Test Generation

Generating valid inputs is a challenging testing problem. Software testing aims to generate inputs that cover program behavior and expose faults. Typical (random) test generation approaches often produce *invalid inputs*, such inputs require further *input repair* before they can be processed by the intended program. To generate valid inputs and avoid input repair, test generation methods need to produce valid inputs. Besides, executing the program logic requires *valid inputs* that pass the input validation step of the program, since most programs expect *structured inputs* that meet specific input specifications (e.g. JSON and JavaScript).

Grammar-based test generation addresses this problem by using input grammars to guide the generation of valid inputs. This section highlights the state of the art in (grammar-based) test generation. This lays the foundation for our work on generating inputs that are (dis)similar to common inputs in the wild (*see Chapter 6*). In this dissertation, we apply probabilistic grammars to generate valid software tests. Thus, we also present the background on the interplay of grammar-based testing and probabilistic grammars.

Generating software tests.

The aim of *software test generation* is to find a sample of inputs that induce executions that sufficiently cover the possible behaviors of the program—including undesired behavior. Modern software test generation relies, as surveyed by Anand et al. [66] on *symbolic code analysis* to solve the path conditions leading to uncovered code [67, 68, 69, 70, 71, 72, 73, 74], *search-based approaches* to systematically evolve a population of inputs towards the desired goal [75, 76, 77, 78], random inputs to programs and functions [79, 80] or a combination of these techniques [81, 82, 83, 84, 85]. Additionally, machine learning techniques can also be applied to create test sequences [86, 87]. All these approaches have in common that they do not require an additional model or annotations to constrain the set of generated inputs; this makes them very versatile, but brings the risk of producing false alarms—failing executions that cannot be obtained through legal inputs.

Grammar-based test generation.

The usage of grammars as *producers* was introduced in 1970 by Hanford in his *syntax machine* [88]. Such producers are mainly used for testing compilers and interpreters: CSmith [89] produces syntactically correct C programs, and LANGFUZZ [90] uses a JavaScript grammar to parse, recombine, and mutate existing inputs while maintaining most of the syntactic validity. GENA [91, 92] uses standard symbolic grammars to produce test cases and only applies stochastic annotation during the derivation process to distribute the test cases and to limit recursions and derivation depth. Grammar-based white-box fuzzing [93] combines grammar-based fuzzing with symbolic testing and is now available as a service from Microsoft. As these techniques operate with system inputs, any failure reported is a true failure—there are no false alarms. None of the above approaches use probabilistic grammars, though.

Probabilistic grammars.

The foundations of probabilistic grammars date back to the earliest works of Chomsky [94]. The concept has seen several interactions and generalizations with physics and statistics [95]. Probabilistic grammars are frequently used to analyze ambiguous data sequences—in computational linguistics [96] to analyze natural language, and in biochemistry [97] to model and parse macromolecules such as DNA, RNA, or protein sequences. Probabilistic grammars have also been used to model and produce input data for specific domains, such as 3D scenes [98] or processor instructions [99].

The usage of probabilistic grammars for test generation seems rather straightforward, but is still uncommon. The *Geno* test generator for .NET programs by Lämmel and Schulte [100] allowed users to specify probabilities for individual production rules. Swarm testing [101, 102] uses statistics and a variation of random testing to generate tests that deliberately targets or omits features of interest. The approaches by Poulding et al. [103, 104] use stochastic context-free grammar for statistical testing. The goal of this work is to correctly imitate the operational profile and consequently the generated test cases are similar to what one would expect during normal operation of the system. The test case generation [105, 106] and failure reproduction [107] approaches by Kifetew et al. combine probabilistic grammars with a search-based testing approach. The results [106] show that the combination produces a large percentage of correct inputs and, based on the fitness function, produces a high-branch coverage. In particular, StGP [105] learns stochastic grammars from sample inputs. The authors found that grammar learning from sample inputs improved code coverage, especially for complex programs, the goal of StGP is to evolve and mutate learned grammars to improve code coverage.

In this dissertation, we present a grammar-based testing approach to generate (dis)similar inputs (*see Chapter 6*). In contrast to our approach, the current state-of-the-art approaches either do not learn probabilistic grammars from (real-world) sample inputs or are incapable of generating inputs that are (dis)similar to common inputs in the wild.

Debugging in Practice: An Empirical Study

This chapter is taken, directly or with minor modifications, from our 2017 ICSE paper *How Developers Debug Software—The DBGBENCH Dataset* [108] and our 2017 ESEC/FSE paper *Where is the bug and how is it fixed? an experiment with practitioners* [109]. My contribution in this work is as follows: (I) original idea; (II) partial implementation; (III) evaluation.

“The fundamental principle of science, the definition almost, is this:
the sole test of the validity of any idea is experiment.”
— Richard P. Feynman

3.1 Introduction

Research has produced a multitude of automated approaches for fault localization, debugging, and repair. A recent survey of Wong et al. [39] cites more than 400 publications on fault localization techniques. Hundreds of approaches have also been proposed for automatic program repair [3]. Besides, several benchmarks have become available for the *empirical evaluation* of such approaches. For instance, COREBENCH [14] and Defects4J [110] contain a large number of real errors for C and Java, together with developer-provided test suites and bug fixes. Using such benchmarks, researchers can make *empirical claims* about the efficacy of their tools and techniques. For instance, an effective fault localization technique would rank very high a statement that was changed in the bug fix [39]. The assumption is that practitioners would identify the same statement as *the* fault. Likewise, an effective auto-generated bug fix would pass all test cases [33], under the assumption that practitioners would accept such fixes.

Unfortunately, debugging is not that simple, particularly not for humans. Recently, several research assumptions have turned out to be unrealistic [11, 12, 13, 14, 15]. In empirical studies, the human factor is naturally somewhat left behind. For instance, despite the hundreds of proposed fault localization techniques [39], most practitioners have *never* used an automated fault localization tool [19]. Why is that, and do we really understand how our tools can address the real-world debugging needs of software engineering professionals at work?

Do proposed debugging approaches relate to the way practitioners actually locate, understand, and fix bugs? Given the complexity of the debugging process, one might assume that it would be standard practice to evaluate novel techniques by means of user studies [111]: Does the tool fit into the process? Does it provide value? How? Yet, how humans actually debug is still not really well explored. Between 1981 and 2010, Parnin and Orso [112] identified only a handful of articles that presented the results of a user study—none of which involved actual practitioners *and* real errors. Since the end of 2010 till 2016, we could identify only *three (3) papers* that evaluated new debugging approaches with actual practitioners and real errors [113, 114, 115].³

To address this problem, we collect empirical evidence on every-day debugging experience of software developers. First, we conducted a retrospective survey on debugging practice with hundreds of developers. Secondly, we conducted an observational study with developers diagnosing and fixing real bugs, from which we collected empirical data from hundreds of hours of debugging sessions. Then, we provide another kind of benchmark — DBGBENCH; one that allows *reality checks* for novel automated debugging and repair techniques.

In our retrospective survey, we examine the state-of-the-practice regarding how developers diagnose and repair bugs in the real world. First, we take stock of the state-of-the-practice by investigating the nature of debugging in practice. Then, we ask what practitioners want that would make debugging easier for them, we analyze their responses and determine the properties of debugging tools and auto-generated patches that practitioners find important to consider adoption. We also identify several reasons in favor and against the automation of bug diagnosis and repair. We then discuss implications of these findings for debugging research and training.

In our observational study, we find the first evidence that debugging *can* actually be automated and is no subjective endeavor. In our experiment, *different* practitioners provide essentially the *same* fault locations and the *same* bug diagnosis for the *same* error. If humans disagreed, how could a machine ever produce the “correct” fault locations, or the “correct” bug diagnosis? Since, there is a consensus on bug diagnoses and patching, we collected the data from this observational study to support the automatic evaluation of debugging and repair tools. We provide these data and findings as a benchmark called DBGBENCH.

Our benchmark — DBGBENCH allows for realistic evaluation of debugging and repair tools. Since participants agree on essential bug features, it is fair to treat their findings as *ground truth*. We have compiled our study data for all 27 bugs into a *benchmark* named DBGBENCH [116], which is the third central contribution of this study. DBGBENCH can be used in *cost-effective user studies* to investigate how debugging time, debugging difficulty, and patch correctness improve with the novel debugging/repair aid. DBGBENCH can be used to evaluate *without a user study* how well novel automated tools perform against professional software developers in the tasks of fault localization, debugging, and repair.

The remainder of this chapter is organized as follows: First, we present a retrospective study investigating the state-of-the-art in debugging practice in Section 3.2. In Section 3.3, we present

³We surveyed the following conference proceedings: ACM International Conference on Software Engineering (ICSE’11–16), ACM SIGSOFT Foundations of Software Engineering (FSE’11–16), ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA’11–16), IEEE/ACM International Conference on Automated Software Engineering (ASE’11–16), and International Conference on Software Testing, Verification and Validation (ICST’11–16) and identified 82 papers on automated debugging or software repair only 11 of which conducted user studies. From these only three (3) involved software engineering professionals *and* real errors.

an observational study evaluating how developers debug and fix real bugs. Section 3.4 describes DBGBENCH— a benchmark for evaluating automated debugging and automated program repair tools. We present the limitations of our work in Section 3.5 and closely related work in Section 3.6. Finally, we close with our interpretation of the results and their implications in Section 3.7.

3.2 Survey

In a survey that ran over 14 months, we collected empirical evidence on the state of debugging in practice from 212 software engineering professionals in 41 countries. We asked questions about their present every-day debugging experience and their expectations and beliefs regarding the future of debugging. We make the questionnaire and response data available on our webpage [116]. Our insights go a long way towards addressing unwarranted beliefs, building tools that solve problems that practitioners actually have, and better educating future software engineering professionals about the pitfalls of manual debugging and about actionable opportunities to make debugging a more efficient and systematic activity. Specifically, we investigate the following about debugging practice, *what tools do practitioners use?*, *what tools do practitioners want?* and *what do practitioners believe about debugging automation?*

This section is structured according to the guidelines for empirical research in software engineering [117]. After this survey introduction, in Section 3.3.1, we introduce the research questions, the population and how it was sampled plus which parameters we measured as *study design*. The collected data is analyzed and presented in Section 3.3.2 as *study results*.

3.2.1 Study Design

The study design describes the population of interest, how it was sampled, and demographic attributes of that sample. The study design also describes the variables of interest and how they are measured. The goal of the study design is to ensure that the design is appropriate for the objectives of the study. Since our study is explorative in nature many questions are qualitative and open-ended. We leverage the standard measures such as the Likert-scale to analyze *qualitative* attributes, and grounded theory, specifically open-card sort, as systematic methodology to analyze the responses to *open-ended* questions. To assess the reliability of the conclusions drawn from the self-reports, we measure inter-rater agreement.

Research Questions

The main objective of the survey is to review the state-of-the-practice in debugging and elicit the expectations and beliefs of software engineering professionals about the automation of their debugging activities. Thus, we pose the following research questions:

RQ1 How do developers debug? How much *time* do practitioners spend trying to reproduce, diagnose, and fix bugs; how *familiar* are they with the source code? Which *debugging tools and techniques* do practitioners use and would they classify their strategy as *systematic or trial-and-error*?

RQ2 What do they want? Given an error and possibly a failing test case, which output do practitioners expect of an *automated bug diagnosis tool* that should aid in understanding how the error comes about? Which criteria do practitioners expect to be satisfied to accept a patch that is generated by an *automated bug fixing tool*?

RQ3 Why do they reject automation? Do practitioners believe that the chain of run-time events leading to the observed error can ever be *explained intuitively* by the push of a button? Do practitioners believe that bugs can ever be *fixed reliably* by the push of a button? *Why* do they believe so?

Our insights go a long way towards addressing unwarranted beliefs, building tools that solve problems that practitioners actually have, and better educating future software engineering professionals about the pitfalls of manual debugging and actionable opportunities of debugging automation. We hope that the results of our survey will inspire more research that is grounded in practice.

Questionnaire and Respondents

Recruitment. We set out to examine broadly all software engineering professionals at any level of experience that are involved in the development and debugging of industrial-scale software products. We designed an online questionnaire and sent the link to developers that shared their email address on the source-code repository platform Github. We also posted the link to several software development user groups at Meetup.com, on freelancer platforms, such as Freelancer, Upwork, and Guru, and on social media platforms as well as professional networks, such as Facebook and LinkedIn. We started three advertisement campaigns in August 2015, March 2016, and July 2016 following which we had the highest response rate lasting for about one month each. We received the first response in August 2015 and the most recent response more than one year later in October 2016.

Questionnaire. The questionnaire takes about 10 minutes to fill and is strictly anonymous.⁴ It begins with a consent form informing respondents about the goals of our study and some basic terminology. We assure that all responses are treated as confidential, and in no case will responses from individual participants be identified; rather, all data will be pooled and published in aggregate form only. Respondents start by answering a series of simple demographic questions about occupation, experience, and level of skill as software developers. After that follows a series of technical questions which investigate our study objectives. The questionnaire is available at <http://bit.do/dbg> and on our webpage [116].

Demographics. The questionnaire was filled by 212 respondents. The *majority*⁵ of respondents are professional software developers from all over the world with seven (7) years or more experience in software development rating their level of skill as advanced or expert. 706 candidates opened the link either directly or from 35 distinct referrer webpages.⁶ Figure 4 shows the 41 home

⁴Participation is strictly anonymous if the respondent so wishes. However, we do offer the opportunity to leave their email address in case they want to be informed about the study outcome or participate in a follow-up hands-on experiment.

⁵Majority here means above the median (i.e., 3rd and 4th quartile).

⁶Statistics are as of 07th Jan'17 and provided at <http://bit.do/dbg->. From 706 distinct IP addresses, 14 are from robots, like the Google web crawler.

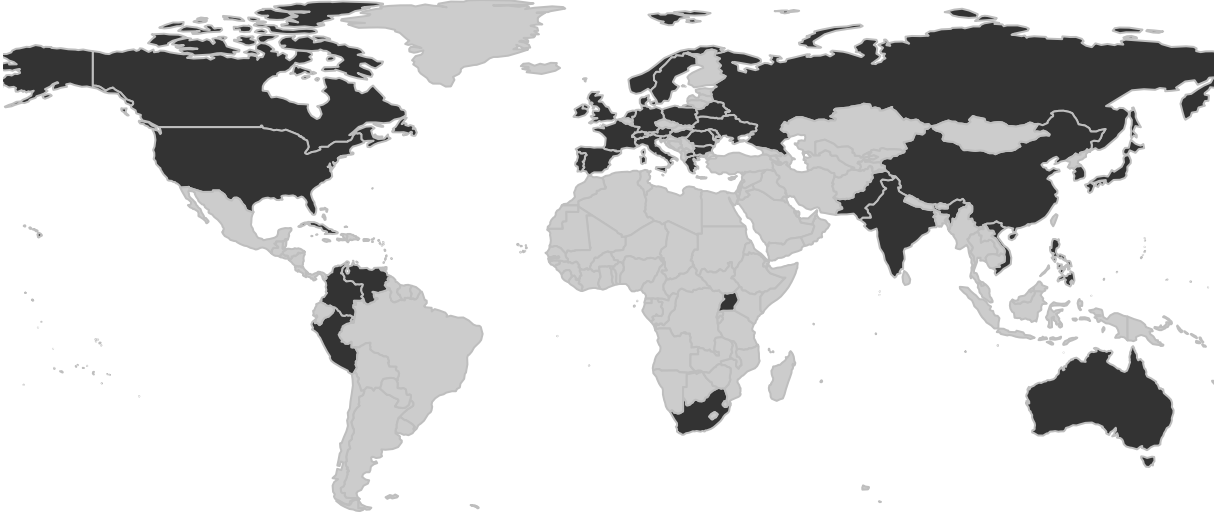


Figure 4: Countries associated with the IP addresses of our survey respondents

countries of the 212 respondents. 116 (55%) respondents categorize themselves as professionals, 47 as students, 32 as researchers, and 6 as professional software tester, the remaining did not specify. 117 (55%) respondents report to have seven or more years of experience, 53 have three to six years, 26 have one to two years, and only 16 have no experience or one year or less. 85 (40%) respondents rate their level of skill as advanced, 35 as expert, 70 as intermediate, and 22 as novice.⁷ We received the first entry in August 2015 and the most recent entry one year later in October 2016. Hence, the survey is not a snapshot taken at a particular point of time but sufficiently long-running to *reduce potential short-term socio-economic bias*.

Variables and Measures

Qualitative Variables. In some cases, we are interested in measuring qualitative properties which may seem hard to quantify. Variables of interest include the *familiarity with the code that is debugged* and the *frequency with which certain tools are used*. To measure these attributes, we utilize the *5-point Likert scale* which is widely used in studies of sociology and psychology [119]. A 5-point Likert scale allows to measure otherwise qualitative properties on a symmetric scale where each item takes a value from 1 to 5 and the distance between each item is assumed to be equal.⁸ For instance, we ask: “*Generally speaking, how familiar are you with the code that you are debugging?*” and provide the following five options:

- () Not at all familiar = *Likert-value 1*
- () Slightly familiar = *Likert-value 2*
- () Moderately familiar = *Likert-value 3*
- () Very familiar = *Likert-value 4*
- () Extremely familiar = *Likert-value 5*

⁷Note that self-assessment of level of skill should always be taken with a grain of salt (cf. Dunning-Kruger effect [118]).

⁸However, the Likert-scale is robust to violations of the equal distance assumption: Even with larger distortions of perceived distances between items (e.g., slightly vs. moderately familiar), Likert performs close to where intervals are perceived equal [120].

An average code familiarity of 4.7 would indicate that most respondents report to be *very to extremely familiar* with the code and only very few report to be *not at all familiar*. However, conclusions from such data can be trusted only after demonstrating their reliability. To assess the reliability of the resulting findings, we measure inter-rater agreement and specifically Fleiss' kappa κ [121]. If respondents are in complete agreement, then $\kappa = 1$. If there is no agreement other than what would be expected by chance, then $\kappa \leq 0$.

Time. We are interested in the time that practitioners spend debugging, and the time that practitioners spend with each of the three sub-tasks of debugging: bug reproduction, diagnosis, and fixing. We measure debugging time *relative to the work time* and the time spent on each sub-task *relative to the debugging time* spent. We measure time on an *interval scale* from 0% - 100% in ten percent intervals that is finer-grained on either side (five percent intervals).

Tools and Techniques. We ask respondents about the following set of tools and techniques:

- [] Trace-based Debugging (using printing; e.g., `println`, `log4c`)
- [] Interactive/Online Debugging (using breakpoints; e.g., `gdb`, `jdb`)
- [] Post-Mortem/Offline Debugging (using core dumps, stack traces)
- [] Regression Debugging to find faulty changes (e.g., `git bisect`)
- [] Statistical/Spectrum-based Debugging to find faulty statements (e.g., Tarantula)
- [] Program Slicing (e.g., Frama-C, CodeSurfer)
- [] Time Travel or Reversible Debugging (e.g., UndoDB)
- [] Algorithmic or Declarative Debugging (e.g., JavaDD)

Open-ended Questions. Most questions about beliefs and expectations (**RQ2**, **RQ3**) are open-ended in the sense that respondents are asked to fill a text field instead of setting checkmarks on pre-specified answers. This *reduces experimenter bias*. To analyze replies, we borrow a methodology from grounded theory called *coding* [122]. In the context of software engineering [123, 124], this methodology is also referred to as *open card sort* [125]. To derive the general categories from the responses and sort the responses to these categories, we used the following *coding protocol*:

1. Researcher #1 went through all responses and came up with an initial draft of the coding. The *coding* comprised of a set of *labels* that are attached to the responses and a *policy* that specifies when to apply which label.
2. Researchers #1, #2, and #3 *discussed* and amended labels and policy *but not the coding draft* by Researcher #1.
3. Researchers #1, #2 and #3 independently *produced* their own codings according to policy (Researcher #1 repeated with the updated policy).
4. Researchers #1, #2, and #3 *compared* their codings and resolve any contention to produce the final coding.

Notice that the labels had not been predefined but emerged during coding. Independent coding reduces potential bias. The coding allows to quantify the *prevalence* of certain “categories” of responses. The authors spent about half a day in meetings to discuss and amend labels and

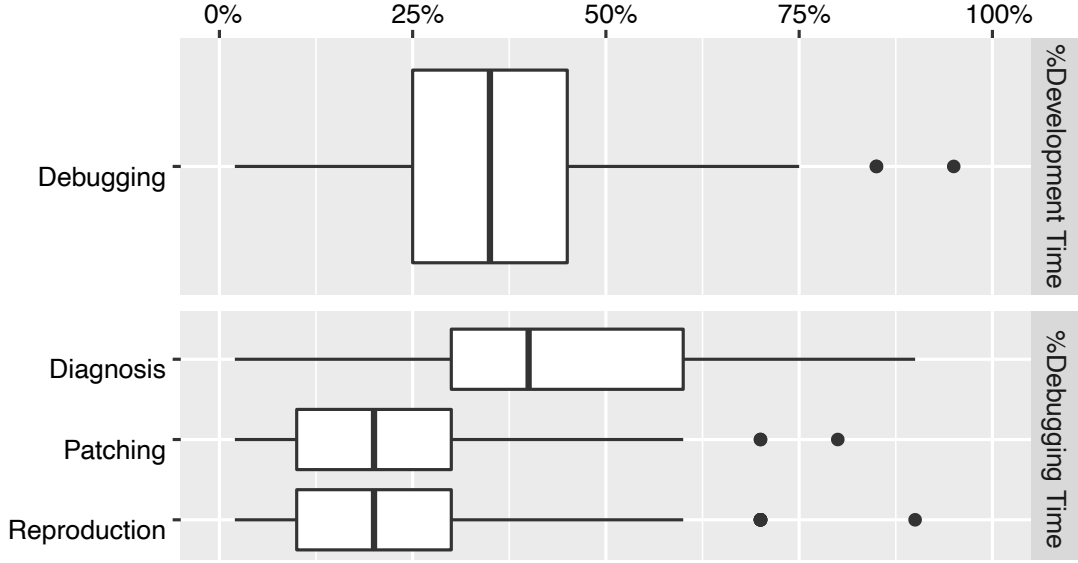


Figure 5: Boxplots of the time spent reproducing, diagnosing, and fixing bugs. The boxplot at the top shows the reported time spent debugging as a percentage of the development time. The boxplots below show the reported time spent on each subtask as a percentage of debugging time.

policy (Step-2) and three days resolving contentions (Step-4) in addition to the actual coding (Step-1&3).

Redundant Questions. Due to the self-reporting nature of a questionnaire it is customary to reduce cognitive bias with an empirical assessment tool called *subject triangulation* [126]. We ask separate questions about the same subject and check the consistency of the replies to assess the validity of our insights. For instance, we ask how familiar the developers are generally with the code they debug on a 5-point Likert scale. Several questions later, we ask how often they debug other peoples code on a 5-point Likert scale. If developers generally feel *slightly* or *moderately familiar*, they should also debug other peoples’ code *moderately* or *very often*.

Bias Mitigation. In summary, we use several empirical devices to mitigate sources of cognitive bias in our self-reporting study. We ask redundant questions for subject triangulation [126]. We avoid leading questions. We ask many open-ended questions, which allows respondents to express themselves more holistically than if we used pre-specified answers. We use standard measures of qualitative attributes [119] and leverage a systematic methodology from grounded theory to analyze the open-ended questions [122]. To assess the reliability of our conclusions from this survey data, we leverage a measure of inter-rater agreement [121].

3.2.2 Study Results

The survey ran over *14 months* and gathered *212 responses*. For the 3392 ratings on all 16 multiple-choice questions, we measure a reasonable, fair agreement amongst the respondents above the level of agreement expected by chance (Fleiss’ $\kappa = 0.301$). This suggests reliability of our conclusions from the self-reported data.

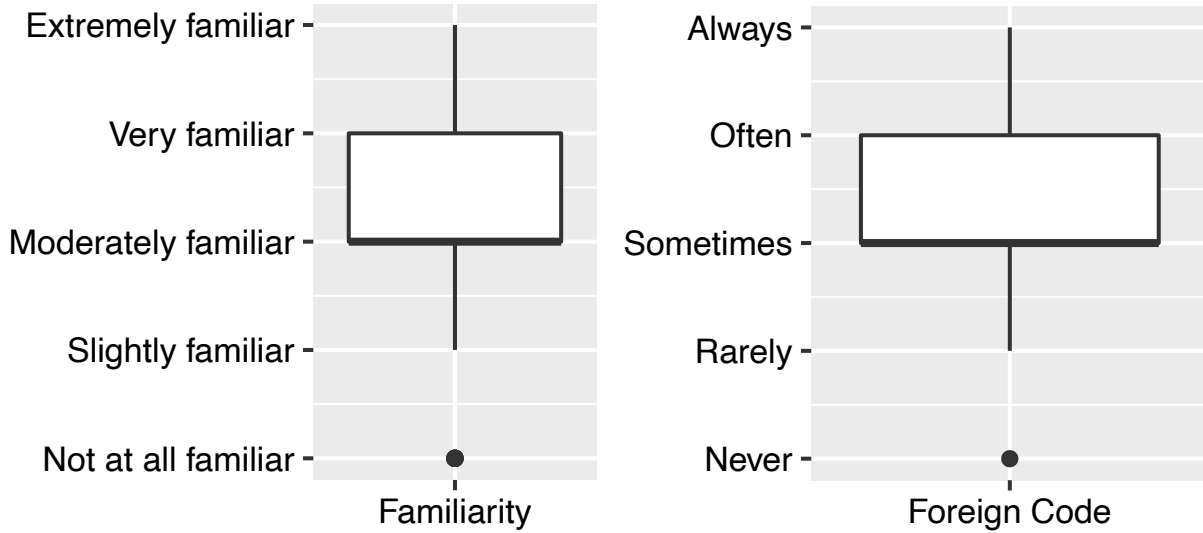


Figure 6: Boxplots of respondent’s familiarity with the code they debug (left) and of the frequency with which they debug other people’s code (right).

RQ1 What is the State of the Practice?

We examine the time spent by developers when debugging, their familiarity to the code the debug and the tools employed during debugging.

Time. *Respondents reported to spend one-third of their working time debugging. Half the debugging time is spent trying to understand how an error comes about (bug diagnosis) while the other half is spent trying to reproduce or patch software errors.* As shown in Figure 5, the middle fifty percent reported to spend between 25% and 45% of their development time with debugging. This confirms the results that were reported by Perscheid et al. [19]. Moreover, we found that during debugging, the middle fifty percent reported to spend between 30% and 60% for bug diagnosis and between 10% and 30% each for bug reproduction and bug fixing.

Familiarity. *Most respondents frequently did not write the software they are debugging. Nevertheless, they feel quite familiar with the code.* As shown in Figure 6, the average familiarity with the code that respondents generally debug is *moderate* (3.3 on the 5-point Likert scale). The middle fifty percent is *moderately* to *very familiar* with the debugged code. We use the frequency with which respondents debug other people’s code to triangulate (i.e., confirm) familiarity [126]. The average frequency is *sometimes* (3.4). The middle fifty percent *sometimes* or *often* debug other people’s code.

Developers spend a third of their time debugging code mostly written by other programmers: 50% of the time is spent on bug diagnosis, and the rest on bug reproduction and patching.

Tools and Techniques. *In practice, debugging is still vastly a manual activity.* As shown in Figure 7, the middle fifty percent *always* or *often* use trace-based debugging (e.g., `println`) or interactive debugging (e.g., `gdb`). They *sometimes* use post-mortem debugging (e.g., inspecting coredump and stack traces). They *rarely* use regression debugging (e.g., `git bisect`). The majority of respondents *never* used any of the remaining choices. Respondents mentioned memory, coverage, and performance profiler and analysis tools, such as `valgrind`, `gcov`, and `gprof` as additional tools which they use, but which are not listed. 60 respondents (30%) admit to trial-and-error versus a

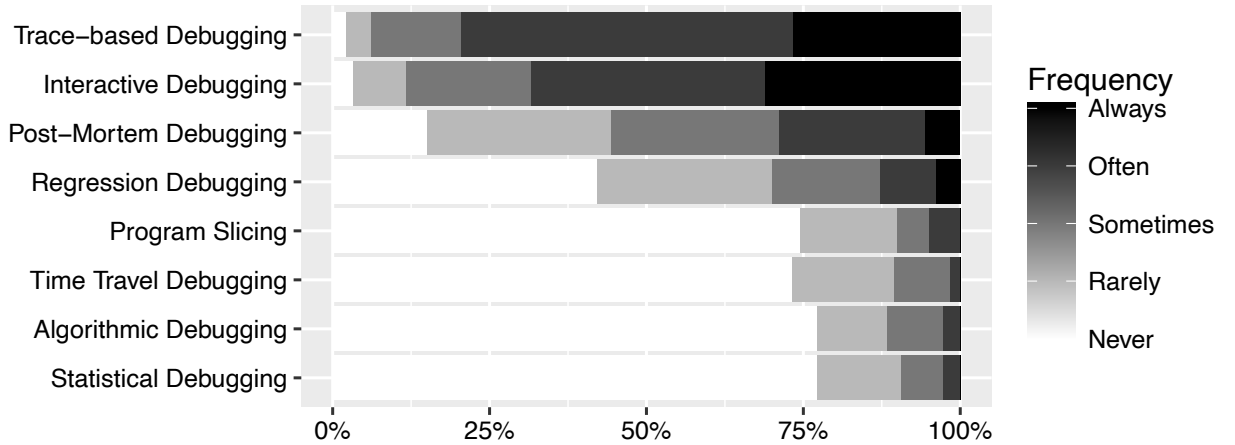


Figure 7: Stacked histograms showing how often respondents use the listed debugging techniques and associated tools. For instance, the first row can be read as follows: About 5% of respondents use trace-based debugging never or rarely, 15% sometimes, 50% often, and 30% always.

more systematic approach.

Debugging is still mostly a manual activity in software practice: trace-based (e.g. `println`) and interactive debugging (e.g. `gdb`) are the most used debugging technique.

Our results for **RQ1** suggest that debugging is still a largely manual and time-consuming process. Many developers find their own style of debugging to be trial-and-error rather than systematic. This is in stark contrast to the many tools and techniques available from many decades of software engineering research on automated debugging, and directly motivates our next research question: What do practitioners want?

RQ2-a How to Automate Bug Diagnosis?

In 127 of 156 valid responses (81%),⁹ practitioners mention they would design automated debugging aids that do what has already been achieved in automated debugging research. For instance, 69 respondents (44%) would like to use tools that point out suspicious statements. However, confirming the observation of Kochar et al. [41], we find that practitioners have information needs that go beyond simple fault localization: 49 of 69 respondents which wanted to be pointed to suspicious statements (71%), were interested in more information, like actual and expected variable values, or an English explanation. Automated repair is an active research topic. We found that 26 of 156 respondents (17%) would output an auto-generated patch as debugging aid.

Figure 8 lists some answers to our open-ended question on the preferred output of an automated diagnosis tool. To establish the categories and quantify the prevalence of each category, we used the *coding protocol* that was discussed in Section 3.3.1. If we did not find any related work that addresses a practitioners' need, the concern is shown in **bold** face. The prevalence of a category in terms of number of valid responses is shown to the left in grey.

In terms of general *program comprehension*, we received 33 valid responses. Developers would design tools that

⁹We marked responses as invalid that were empty or did not answer the question.

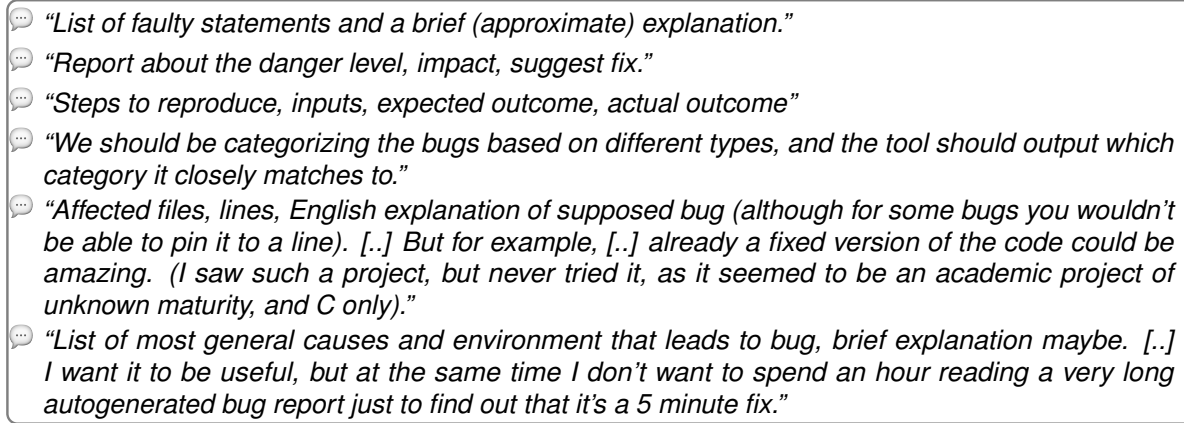


Figure 8: Examples of responses to our Question 16: "If you could design an automated bug diagnosis tool that explains the reproduced bug to you, what would it output?"

- 24 visualize data structures and allow to persist, to restore, and to compare their states [127],
- 8 visualize the value history of a variable [127],
- 5 help at program understanding, generate documentation [128],
- 1 uncover "meaning" of variables and value range [129, 130].

In terms of *automated bug diagnosis*, we received 148 valid responses. Developers themselves would design tools that

- 69 point to suspicious functions or program statements [131],
- 41 **generate an English explanation why the error occurs**,
- 37 print the sequence of executed functions for a failing input (as can be obtained automatically via any online-debugger),
- 26 generate a patch to assist in understanding the error [115] or generate a suggestion where and how to patch the bug [132],
- 21 **report most general environment or conditions under which the bug can be reproduced**,
- 15 **visualize divergence from the *expected* value of a variable**,
- 10 **visualize the range of *expected* values for a given variable**.
- 7 point to the cause-effect chain leading to the symptom [32].

In terms of *explaining and classifying symptoms*, we received 26 valid responses. Developers are interested in tools that

- 17 **highlight the symptoms and side-effects of an error**,
- 10 **classify the error according to its symptom in a category (e.g., if nullpointer deref., suggest check or where to init.)**,
- 2 **evaluate criticality of the symptoms (e.g., security risk)**,
- 1 track allocated resources and where they are allocated or used (as can be obtained via tools like *valgrind* [133]).

Properties of the Tool. *Bug diagnosis tools should enable the practitioner to investigate properties of the program and the failing execution. 27 of 50 valid responses¹⁰ (54%) mention*

¹⁰Invalid responses are empty or do not pertain to properties of the bug diagnosis tool.

-
- Figure 9 shows a box containing four speech bubble icons, each followed by a response to the question 'What do you expect from an automated diagnosis tool?'. The responses are: 1. 'Diagnosis tools should help narrow down the list of possible causes, so I can focus my investigation. [...] Instead of replacing a human, the tool should help the human do their job faster.' 2. 'I'd expect that tool is easy to use, readable, informative and fast.' 3. 'Somehow you need to be able to teach the tool what constitutes a bug and what does not.' 4. A numbered list: '1. Scalability. It should allow me to debug even very large program. 2. The output should be concise, precise and easy to understand. 3. The tool could give some hints to fix the bug if possible.'
- “Diagnosis tools should help narrow down the list of possible causes, so I can focus my investigation. [...] Instead of replacing a human, the tool should help the human do their job faster.”
 - “I’d expect that tool is easy to use, readable, informative and fast.”
 - “Somehow you need to be able to teach the tool what constitutes a bug and what does not.”
 - “1. Scalability. It should allow me to debug even very large program.
2. The output should be concise, precise and easy to understand.
3. The tool could give some hints to fix the bug if possible.”

Figure 9: Examples of responses to our optional Question 24: “What do you expect from an automated diagnosis tool?”

that the diagnosis tool should be able to conduct sophisticated program analysis (beyond fault localization). Moreover, the tool should be efficient, interactive, easy to use, and support the execution and construction of a test suite. Figure 9 shows some example responses. Concretely, practitioners expect a bug diagnosis tool to satisfy the following criteria:

- 27 *Program Analysis*: Conduct analyses, such as slicing, as needed.
- 14 *Efficient*: Be quick, use few resources, scale to large programs.
- 8 *Interactive*: Arrive at the diagnosis with help from the developer.
- 8 *Testing*: Execute test cases and collect code coverage as needed.
- 7 *Easy to use*: Not too complicated or too extensive.
- 2 *Trainable*: Learn from the interaction with the developer.

The most desired functional properties of debuggers are already being provided by the state-of-the-art debugging tools (e.g. program analysis is desired by 54% of respondents).

Two respondents mentioned that a good integration into existing IDEs, such as Eclipse, would be preferable. One would want to integrate it into the existing patch review process if diagnosis indeed was fully automated.

Properties of a Diagnosis. *Most importantly, an auto-generated bug diagnosis should provide all information that is necessary to understand how the bug is coming about. 32 of 50 valid responses¹¹ (64%) mention that a diagnosis should be comprehensive. Moreover, a diagnosis should provide the correct diagnosis in an understandable format.* Concretely, practitioners expect an auto-generated bug diagnosis to satisfy the following criteria:

- 32 *Comprehensive*: Provide all information that is relevant. For instance, fault locations, context, and variable values.
- 16 *Accurate*: Provide only correct information.
- 9 *Understandable*: Provide pertinent information in an intuitive format that is easy to navigate. Text should be comprehensible.
- 3 *Simple*: Provide not more information than necessary.

Most debugging tools do not provide the desired non-functional properties of bug diagnosis: for instance, 64% of respondents desire comprehensive bug diagnosis.

¹¹Invalid responses are empty or don’t mention properties of auto-generated diagnoses.

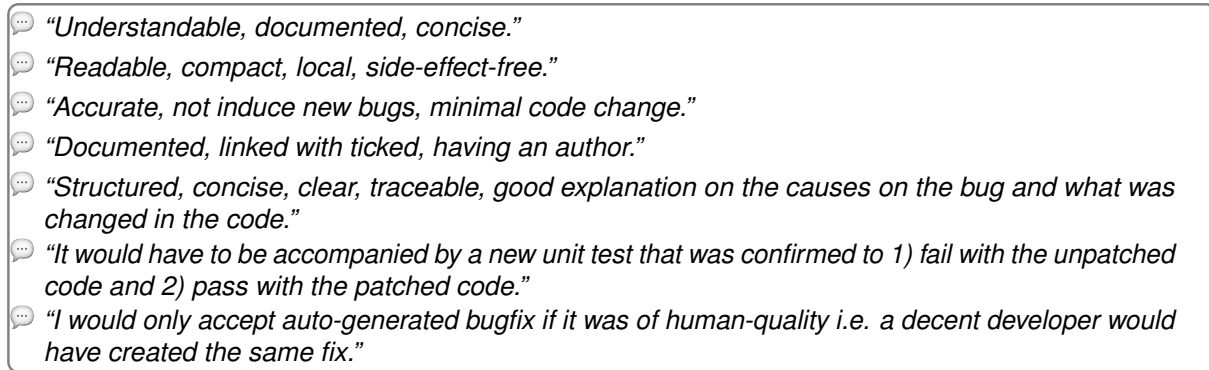


Figure 10: Examples of responses to our Question 15: "If you had to review an (auto-generated) bug fix / software patch, which properties must it have to be acceptable?"

Triangulation. In the questionnaire, we asked two questions about the automation of bug diagnosis. The first asks concretely about the format and content of the auto-generated diagnosis (Figure 8) while the other asks more generally about the expectations from a bug diagnosis tool (Figure 9). Since the questions are open-ended and responses in prose, in many cases, there was some information in the latter about the former and vice versa. We used these responses to triangulate the categories established above and found mostly agreement and no contradictions.

RQ2-b How to Automate Bug Fixing?

Most respondents would accept an auto-generated patch only if it is readable, well-documented, and does not introduce any new bugs. Moreover, an auto-generated patch should be plausible, minimal, and well-structured. It should provide the context such as a reference to the bug report and discuss the potential impact of the patch on other parts in the program. The current problem statement of research in automated program repair is to produce patches that pass all test cases in a given test suite. However, correctness is not the only concern of practitioners. In fact, only two respondents would require the patch to fix the error completely. The main concern is that an auto-generated patch is readable and documents what has been done and why.

For this question, we collected 175 valid responses. Figure 10 shows a few examples. Concretely, practitioners would agree to accept an auto-generated patch if it satisfies the following properties:

- 80 *Be readable*: Self-explanatory, coding standards, variable names.
- 41 *Be documented*: An explanation of what has been done and why. Auto-generate a comprehensive commit log.
- 36 *No new bugs*: No tests producible that pass before but fail now. Auto-generate regression tests.
- 29 *Be plausible*: Passes failing test used during bug reproduction.
- 27 *Provide context*: Links the bug report, patch reviewer, identifies environment where it was tested.
- 24 *Be minimal*: Local to 1 module, small changes, contained impact.
- 22 *Discuss impact*: Explain modules impacted by this patch and how.
- 21 *Be well-structured*: Well-formatted, matches existing style and clean.
- 2 *Fix the bug*: No tests producible that fail because of this error.

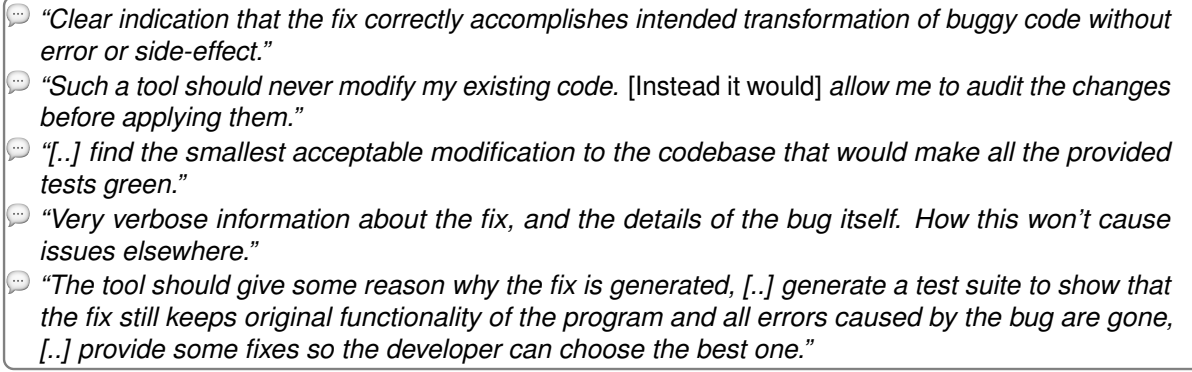


Figure 11: Examples of responses to our Question 25: "What do you expect from an automated bug fixing tool?"

Properties of the Tool. On the one hand, 26 of 64 respondents (41%) would not like the tool to fix a program fully automatically and instead intervene whenever necessary or select from a ranked choice of patches. On the other hand, 19 respondents (30%) say that such tool would need to fix a bug on its own. However, *most respondents (63%) would like the tool to provide a proof of correctness (22), an explanation of how the bug is fixed (13), or at least guarantee correctness (18).*¹²

For this optional question, we collected 64 valid responses. A few examples are shown in Figure 11. Concretely, practitioners would agree to use an auto-repair tool if it satisfies the following criteria. The tool should be:

- 26 *Interactive*: Arrive at the correct fix with help from developer. For instance, produce several patches rank in order of quality.
- 19 *Fully automated*: Produce and apply bug fix fully automatically.
- 22 *Provide proof of correctness*: Generate regression test cases. Show that error was really fixed and no new errors are introduced.
- 18 *Correct*: Does not produce incorrect patches.
- 13 *Provide rationale*: Explain the bug and how it is fixed now.
- 7 *Decide degree of automation based on bug type*: Some errors may need almost no intervention and can be auto-corrected.
- 4 *Be efficient*: Be quick, use few resources, scale to large programs.
- 1 *Be open-source*: So that developers can inspect how it works.

Beyond patch correctness, the most desirable properties of repair tools (e.g. fix explanation and correctness guarantee) are not provided by the current state-of-the-art repair tools.

Triangulation. In the questionnaire, we asked two questions about the automation of bug fixing. The first asks concretely about properties of auto-generated patches that are required to accept the patch (Figure 10) while the other asks more generally about the expectations from an automated bug fixing tool (Figure 11). We used these responses to triangulate the categories established above and found mostly agreement and no contradictions.

¹²Note that a response can be assigned multiple labels.

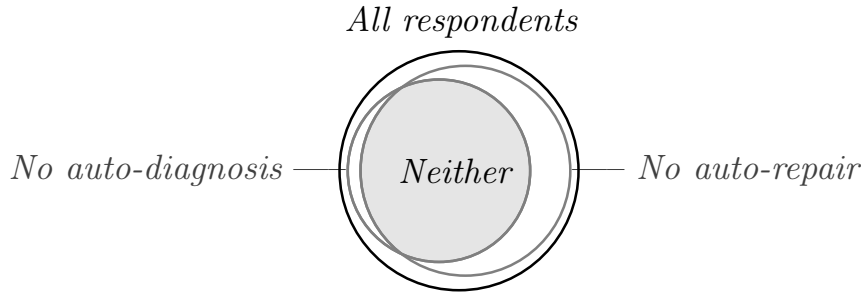


Figure 12: Venn Diagram showing the set of respondents which oppose the full automation of either bug diagnosis (no auto-diagnosis) or bug fixing (no auto-repair).

RQ3 Why Accept/Reject Automated Debugging?

Most surveyed practitioners do not believe that debugging will ever be fully automated. The Venn diagram in Figure 12 provides more details. 116 respondents (55%) believe that neither bug diagnosis nor program repair will ever be fully automated. 165 (78%) believe that bugs will *never* be fixed reliably by a machine while 124 respondents (58%) believe that bugs will *never* be explained intuitively.

Points in favor of Automation. 33 of 96 proponents¹³ provide a rationale in favour of the automation of bug diagnosis, repair, or both. The most prevalent responses cite recent advances in artificial intelligence as reason why debugging might be fully automated in the future. Others mention that existing techniques do not sufficiently leverage information that is available, in the program itself or in lessons-learned from earlier debugging sessions.

Concretely, 33 practitioners explained that debugging might be automated at some point in the future for the following reasons:

- 14 *Advances in AI*: Given the progress of Artificial Intelligence (AI) in other areas, it may soon also be able to auto-repair programs.
- 9 *Information from program*: Stack traces and source code can be mined via program analysis for information on a bug fix.
- 5 *Information from history*: Repositories of human-generated bug reports and patches can be mined for information on a patch.
- 3 *Information from specifications*: Adequate test suites or formal specifications should contain enough information to fix a bug.
- 2 *Advances in Self-Healing*: In future, a software system may test and heal itself and automatically recover from runtime errors.
- 2 *Advances in Optimisation Algorithms*: Genetic algorithms have shown promise, taking inspiration from natural evolution.
- 2 *Advances in Program Synthesis*: If programs are generated from specifications, then bugs are fixed by changing the specification.

A third of developers (34%) believe in the (future) automation of debugging or repair because of advances in artificial intelligence (AI) and program analysis.

¹³We call as *proponent* a respondent who believes that *diagnosis*, *repair*, or *both* will be automated at some point of time in the future (white area in Figure 12). In other words, even if the respondent believes that one task, e.g., diagnosis, will *never* be automated, she is still classified as a proponent. Only 32 of 212 respondents (15%) believe that both, diagnosis and patching may be automated in the future (13 provide a rationale).

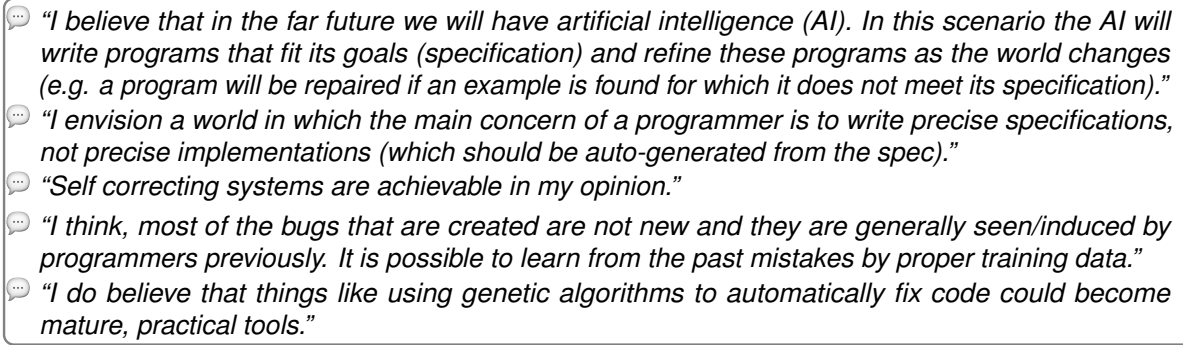
- 
- “I believe that in the far future we will have artificial intelligence (AI). In this scenario the AI will write programs that fit its goals (specification) and refine these programs as the world changes (e.g. a program will be repaired if an example is found for which it does not meet its specification).”
- “I envision a world in which the main concern of a programmer is to write precise specifications, not precise implementations (which should be auto-generated from the spec).”
- “Self correcting systems are achievable in my opinion.”
- “I think, most of the bugs that are created are not new and they are generally seen/induced by programmers previously. It is possible to learn from the past mistakes by proper training data.”
- “I do believe that things like using genetic algorithms to automatically fix code could become mature, practical tools.”

Figure 13: Responses in favour of debugging automation

Points Against Automation. 51 of 116 opponents¹⁴ provide a rationale against the automation of both – diagnosis and repair. The most prevalent reason cited is that some errors are just too difficult to explain even by humans. Only if there is consensus among developers about the explanation of how a bug comes about, there can be an auto-generated bug diagnosis that is agreeable to most developers. However, 156 respondents (72%) believe that there is no single explanation for a bug. Meaning, practitioners believe the cause of a bug is subjective.

The problem of the difficulty of correct bug diagnosis is exacerbated when it is not clear whether some behavior is a bug or a feature and the system is too complex that a machine cannot have complete knowledge of all factors. Moreover, respondents believe that a machine lacks the experience and ingenuity of the human, specifically in the presence of existing bad coding practices.

Concretely, practitioners believe that debugging will never be automated for the following reasons.

- 22 *Complexity of diagnosis:* Some errors are difficult to explain even for the human. How should a machine be ever able to?
- 19 *Absence of specification:* Without a complete specification, a machine is unable to distinguish between bug and feature.
- 17 *Complexity of system:* Bugs may result from complex interaction with hardware. Halting problem prevents reliable analysis.
- 13 *Human Factor:* A machine cannot utilize human experience or ingenuity, or human understanding of the greater context.
- 8 *Difficulty to judge patch quality:* There may be several possible fixes. Which one is the best? According to which criteria?
- 7 *Existing Code Smells:* If the program already incorporates coding bad practices, so will the auto-generated patch.
- 5 *Difficulty to judge patch correctness:* How can a machine know the difference between bug and feature if the human does not even know? How to determine whether the error is really fixed?
- 3 *Difference in-field vs. in-house:* The environment of the auto-repair tool may be different from the user’s environment.
- 2 *Complexity of patch:* Some errors are difficult to patch even for a human. Bug fixes may span several different modules.

¹⁴We call as *opponent* a respondent who believes that *neither diagnosis nor repair* will ever be automated in the future (gray area in Figure 12).

Two-fifth of developers (41%) are against the automation of debugging or repair, due to the complexity of bugs or systems, and the absence of (complete) specification.

Interestingly both, the lack and the presence of specifications is cited as reasons against and in favour of debugging automation, respectively. The *opponents* argue that current software systems lack any formal specification which prevents a machine from distinguishing software bugs from features. Formally, this was dubbed the *oracle problem* [134, 135]. Just recently, Elbaum and Rosenblum called for research on testing techniques that work in the presence of uncertainty [136]. However, the *proponents* argue that future software systems are fully generated from specifications. In other words, any bug in the program originates either in the program synthesizer or in the specification. Assuming the latter, the bug is fixed simply by changing the program specification. For instance, an emerging field in program synthesis is *inductive programming* which allows to generate programs from examples [137]. When the developer finds an input that exposes what she perceives to be a bug, the program is fixed when the input is passed as counter-example to the inductive program synthesizer.

Result Discussion

In the following, we discuss a selection of insights collected from this retrospective survey.

What practitioners use. In the first part of the survey, we find that debugging in practice is still a largely manual and time-consuming process. Developers report to spend half their working time debugging source code which they are often not very familiar with. Among the regularly used techniques are trace-based debugging (i.e., `println`) and interactive debugging (i.e., `gdb`). Most respondents have *never* used any automated techniques, such as program slicing, automated fault localization, or algorithmic debugging. After successfully reproducing a reported bug, developers spend most of their time trying to understand the chain of runtime actions leading to bug (bug diagnosis). Once the bug is understood, the process of manual patch generation is typically faster (bug fixing). Most practitioners ensure that a bug has really been fixed by re-executing the failing test case or re-executing the existing regression test suite. Many practitioners find their own style of debugging to be trial-and-error rather than systematic.

What practitioners want. In the second part of the survey, we identify the pertinent properties of practical bug diagnosis and repair assistants. In terms of *automated bug diagnosis*, practitioners are looking for a tool that produces simple, yet comprehensive reports, that explain why the error occurs, that are easy to navigate and to understand. Beyond automatic fault localization, a bug diagnosis assistant should learn to distinguish actual from expected values, find out under which most general circumstances the bug can be reproduced, and determine the side-effects of an error. In terms of *automated program repair*, the most common problem statement in research is to produce a patch that passes all failing test cases [33, 35]. While the auto-generated patch is often plausible (and pass the failing tests), it might not be correct (and pass the code review) [138]. However, beyond patch correctness practitioners are interested in properties such as readability or impact.

What practitioners believe. In the third part of the survey, we find that *most practitioners*

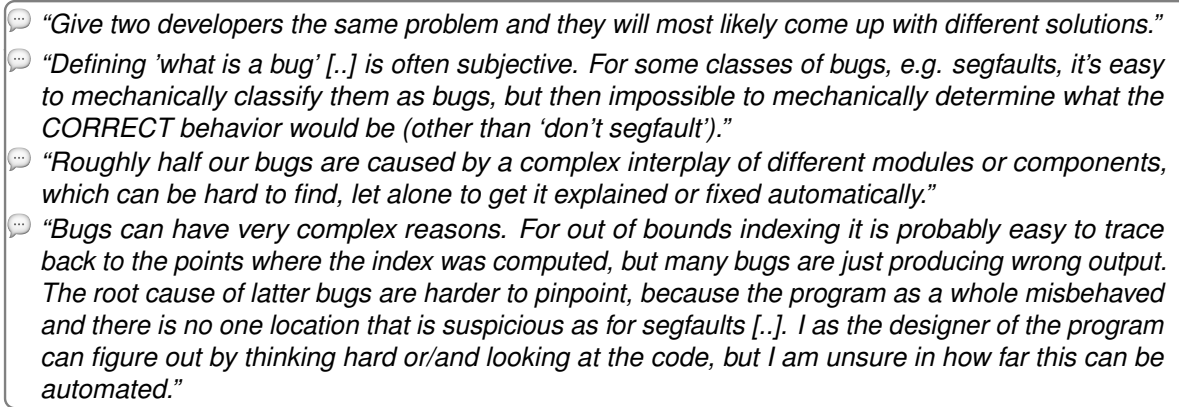


Figure 14: Responses against debugging automation.

reject the notion that diagnosis or repair will ever be fully automated. The most cited reason is that *some errors are difficult to understand even by a human*, specifically when the system is too complex (i.e., a machine cannot have access to all relevant information) and when the correct behavior is unspecified (i.e., a machine cannot distinguish bug from feature). However, practitioners also provide plenty of reasons why at least some debugging tasks might be fully automated at some point in the future. Apart from advances in artificial intelligence, practitioners believe that not every bug is new and common mistakes are repeated, which allows machines to *learn from history*. Practitioners also believe that program analysis will improve and allow to mine more relevant information from the program itself.

At the end of this chapter, we discuss how the community can address the expectations and beliefs of practitioners and provide actionable suggestions for researchers and educators (*see ??*). We hope that the results of our survey will inspire more research in debugging that is grounded in practice.

3.3 Observational Study

In this section, we collect empirical data on the debugging process of developers; we shed light on the *entire debugging process*. Specifically, we investigate how debugging time, difficulty, and strategies vary across practitioners and types of errors. Using 27 real bugs from COREBENCH [14], we asked 12 software engineering professionals from 6 countries to debug and repair software errors. For our benchmark, we elicit which fault locations, explanations, and patches *practitioners* produce.

Participants *received* the following for each error

- a small but succinct *bug report*,
- the buggy *source code* and executable, and
- a *test case* that fails because of this error.

We *asked* participants

- to point out the buggy statements (*fault localization*),
- to explain how the error comes about (*bug diagnosis*), and
- to develop a patch (*bug fixing*).

We *recorded* for each error

- their *confidence* in the correctness of their diagnosis/patch,
- the *steps* taken, the *tools* and *strategies* used, and
- the *time* taken and *difficulty* perceived in both tasks.

We *analyzed* this data and

- derived for each error important *fault locations* and a diagnosis
- evaluated the correctness of each submitted *patch*, and
- provide *new test cases* that fail for incorrect patches.

Thus, we obtained essential data from over 300 individual debugging sessions which can be sliced by developer or by bug, which can serve as reality check for automated debugging and repair tools.

3.3.1 Study Design

The study design discusses our recruitment strategy, the objects and infrastructure, and the variables that we modified and observed in our experiment. The goal of the study design is to ensure that the design is appropriate for the objectives of the study. We follow the canonical design for controlled experiments in software engineering with human participants as recommended by Ko et al. [139].

Research Questions

The main objective of the experiment is to collect empirical data on the debugging process of developers and construct a benchmark that allows to evaluate automated fault localization, bug diagnosis, and software repair techniques based on the judgment of actual professional software developers. We also study the various aspects of debugging in practice and opportunities to automate diagnosis and repair guided by the following *research questions*.

RQ1 Time and Difficulty. Given an error, how much time do developers spend understanding and explaining the error, and how much time patching it? How difficult do they perceive the tasks of bug diagnosis and patch generation?

RQ2 Fault Locations and Patches. Which statements do developers localize as faulty? How are the fault locations distributed across the program? How many of the provided patches are plausible? How many are correct?

RQ3 Diagnosis Strategies. Which strategies do developers employ to understand the runtime actions leading to the error?

RQ4 Repair Ingredients. What are the pertinent building blocks of a correct repair? How complex are the provided patches?

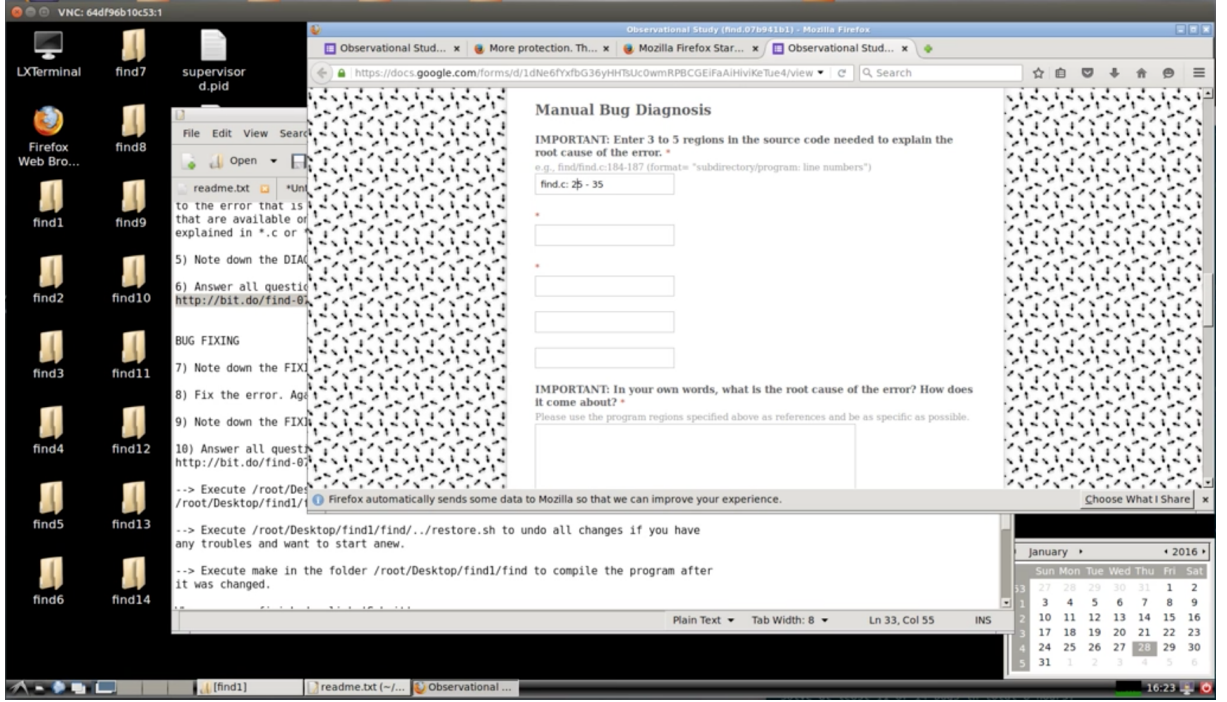


Figure 15: Screenshot of the provided virtual environment.

RQ5 Consensus on Debugging. Is there a consensus among developers during fault localization, bug diagnosis and bug fixing?

RQ6 Debugging Automation. Can debugging be automated? Do developers believe that the diagnosis or repair for an error will ever be automated and why?

Objects and Infrastructure

The *objects* under study are 27 real software errors systematically extracted from the bug reports and commit logs of two GNU open-source C projects (find and grep). The *infrastructure* is a lightweight Docker container that can quickly be installed remotely on any host OS [140]. The errors, test cases, bug reports, source code, and the complete Docker infrastructure is available for download [116].

The objects originate from a larger error benchmark called COREBENCH [14]. Errors were systematically extracted from the 10,000 most recent commits and the bug reports in four projects. Find and grep are well-known, well-maintained, and widely-deployed open-source C programs with a codebase of *17k* and *19k LoC*, respectively. For each error, we provide a failing test case, a simplified bug report, and a large regression test suite (see Figure 26-a). We chose two subjects out of the four available to limit the time a participant spends in our study to a maximum of three working days and to help participants to get accustomed to at most two code bases.

To conduct the study remotely and in an unsupervised manner, we developed a virtual environment based on Docker [140, 116]. The *virtual environment* is a lightweight Docker image with an Ubuntu 14.2 Guest OS containing a folder for each buggy version of either grep or find (27 in total). A script generates the *participant ID* for the responses by a participant. This ensures anonymity and prevents us from establishing the identity of any specific participant. At the same time we can anonymously attribute a response for a different error to the same

participant to measure, for instance, how code familiarity increases over time. The same script does some *folder scrambling* to randomize the order in which participants debug the provided errors: The first error for one participant might be the last error for another. The scrambling controls for learning effects. For instance, if every participant would start with the same error, this error might incorrectly be classified as very difficult. The *docker image* contains the most common development and debugging tools for C, including `gdb`, `vim`, and Eclipse. Participants were encouraged to install their own tools and copy the created folders onto their own machine. A screenshot of the docker image desktop is shown in Figure 15.

Pilot Studies: Researchers and Students

Ko et al. [139] note that the design of a study with human participants is necessarily an iterative process. Therefore, a critical step in preparing an experiment is to run pilot studies. Hence, we first evaluated our design and infrastructure in a *sandbox pilot* where we, the researchers, were the participants. This allowed us to quickly catch the most obvious of problems at no extra cost. Thereupon, we sought to recruit several *graduate students* from Saarland University for the *pilot study*. We advertised the study in lectures, pasted posters on public bulletin boards, and sent emails to potentially interested students. From 10 interested students, we selected five (5) that consider their own level of skill in the programming with C as *advanced or experts*.¹⁵ We conducted the pilot study as *supervised, observational study* in-house, in our computer lab. After filling the consent form and answering demographic questions, we introduced the errors and infrastructure in a small *hands-on tutorial* that lasted about 30 minutes. Then, the students had eight (8) hours, including a one hour lunch break, to debug as many errors as they could. We recorded the screen of each student using a *screen capturing* tool. Independent of the outcome, all participants received EUR 50 as monetary compensation. While *none* of the data collected in the pilot studies was used for the final results, the pilot studies helped us to improve our study design in several ways:

- 1) **No Students.** For the main study, we would use only software engineering professionals. In seven hours our student participants submitted only *a sum total* of five patches. On average, a student fixed one (1) error in eight (8) hours (while in the main study a professional fixed 27 errors in 21.5 hours). The feedback from students was that they under-estimated the required effort and over-estimated their own level of skill.
- 2) **No Screen Capturing.** The video of only a single participant would take several Gigabyte of storage and it needs to be transferred online to a central storage. This was deemed not viable.
- 3) **Good Infrastructure.** The setup, infrastructure, and training material was deemed appropriate.

¹⁵Note that self-assessment of level of skill should always be taken with a grain of salt (cf. Dunning-Kruger effect [118]).

1. **How difficult was it to understand the runtime actions leading to the error?**
(Not at all difficult, Slightly difficult, Moderately difficult, Very difficult, Extremely difficult)
2. **Which tools did you use to understand the runtime actions leading to the error?**
[Textbox]
3. **How much time did you spend understanding the runtime actions leading to the error?**
(1 minute or less, 2–5 minutes, 5–10 minutes, 10–20 minutes, ..., 50–60 minutes, 60 minutes or more)
4. **Enter 3 to 5 code regions needed to explain the root cause of the error.**
[Textbox 1], [Textbox 2], [Textbox 3], [Textbox 4], [Textbox 5]
5. **What is the root cause of the error? How does it come about?**
[Textbox]
6. **How confident are you about the correctness of your explanation?**
(Not at all confident, Slightly confident, Moderately confident, Very confident, Extremely confident)
7. **If you could not explain the error, what prevented you from doing so?**
[Textbox]
8. **Which concrete steps did you take to understand the runtime actions?**
[Textbox]
9. **Do you believe that the root cause of the error can be explained intuitively by the push of a button?**
Yes, in principle a tool might be able to explain this error. No, there will never be a tool that can explain this error.
10. **Why do you (not) believe so?**
[Textbox]

Figure 16: Questions on the fault locations and bug diagnosis

Main Study: Software Professionals

We make available the training material, the virtual infrastructure, the questionnaire that was provided for each error, the collected data [116]. The *experiment procedure* specifies the concrete steps a participant follows from the beginning of the experiment to its end.

Recruitment. We recruited participants from the respondents of our retrospective survey in Section 3.2. The survey asks general questions about debugging in practice after which developers have the option to sign up for the experiment. We sent the link to more than 2,000 developers on Github and posted the link to 20 software development usergroups at Meetup.com, on six (6) freelancer platforms, including Freelancer, Upwork, and Guru, and on social as well as professional networks, such as Facebook and LinkedIn. The job postings on the freelancer platforms were the most effective recruitment strategy. We started three advertisement campaigns in Aug’15, Mar’16, and July’16 following which we had the highest response rate lasting for about one month each. We received the first response in Aug’15 and the most recent response more than one year later in Oct’16. In total, we received 212 responses out of which 143 indicated an interest in participating in the experiment.

Selection. We selected and invited 89 professional software engineers based on their experience with C programming. However, in the two years of recruitment only 12 participants actually entered and completed the experiment. There are several reasons for the high attrition rate. Interested candidates changed their mind in the time until we sent out the invitation, when they understood the extent of the experiments (2–3 working days), or when they received the remote infrastructure and understood the difficulty of the experiment (17k + 19k unfamiliar lines of code).

Demographics. The final participants were *one researcher* and *eleven professional software engineers* from six countries (Russia, India, Slovenia, Spain, Canada, and Ukraine). All professionals had profiles with Upwork and at least 90% success rate in previous jobs.

Training. Before starting with the study, we asked participants to set up the Docker image

12. **How difficult was it to fix the error?**
(*Not at all difficult, Slightly difficult, Moderately difficult, Very difficult, Extremely difficult*)
 13. **How much time did you spend fixing the error?**
(*1 minute or less, 2–5 minutes, 5–10 minutes, 10–20 minutes, . . . , 50–60 minutes, 60 minutes or more*)
 14. **IMPORTANT: Copy & paste the generated patch here.**
[Textbox]
 15. **In a few words and on a high level, what did you change to fix for the error?**
[Textbox]
 16. **How confident are you about the correctness of your fix?**
(*Not at all confident, Slightly confident, Moderately confident, Very confident, Extremely confident*)
 17. **In a few words, how did you make sure this is a good fix?**
[Textbox]
 18. **If you could not fix the bug, what prevented you from doing so?**
[Textbox]
 19. **Do you believe that this error can be fixed reliably by the push of a button?**
Yes, in principle a tool could fix this error reliably. No, there will never be a tool that can fix this error reliably.
 20. **Why do you (not) believe so?**
[Textbox]

Figure 17: Questions on generating the software patch

and get familiar with the infrastructure. We made available *1 readme, 34 slides, and 10 tutorial videos* (~2.5 minutes each) that explain the goals of our study and provide details about subjects, infrastructure, and experimental procedure. Participants could watch the slides and the tutorial videos at their own pace. The training materials are available [116]. Moreover, we informed them that they could contact us via Email in case of problems. We provided technical support whenever needed.

Tasks. After getting familiar with the infrastructure and the programs, participants chose a folder containing the first buggy version to debug. This folder contains a link to the questionnaire that they are supposed to fill in relation with the current buggy version. The text field containing the *participant's ID* is set automatically. The questionnaire contains the technical questions, is made available [116], and is discussed in Section 3.3.1 in more details. We asked each participant to spend approximately 45 minutes per error in order to remain within a 20 hour time frame. From the pilot study, we learned that incentive is important. So, we asked them to fix at least 80% of the errors in one project (e.g., `grep`) before being able to proceed to the next project (e.g., `find`).

Debriefing. After the experiment, participants were debriefed and compensated. We explained how the data is used and why our research is important. Participants would fill a final questionnaire to provide general feedback on experiment design and infrastructure. For instance, participants point out that sometimes it was difficult to properly distinguish time spent on diagnosis from time spent on fixing. Overall, the participants enjoyed the experiment and solving many small challenges in a limited time.

Compensation. It is always difficult to determine the appropriate amount for monetary compensation. Some guidelines [141] recommend the average hourly rate for professionals, the rationale being that professionals in that field cannot or will not participate without pay for work-time lost. Assuming 20 working hours and an hourly rate of USD 27, each participant received USD 540 in compensation for their time and efforts. The modalities were formally handled via Upwork [142].

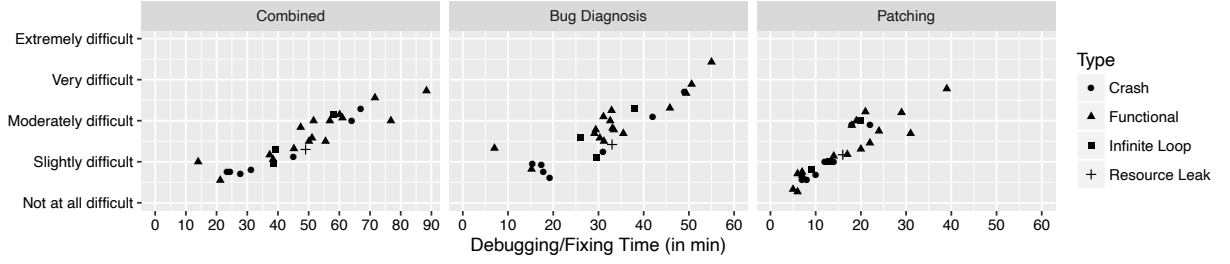


Figure 18: Relationship between average time spent and difficulty perceived for patching and diagnosing a bug. Each point is one of 27 bugs, the shape of which determines its bug type (i.e., crash, functional error, infinite loop, or resource leak).

Variables and Measures

The main objective of this study is to collect the fault locations, bug diagnoses, and software patches that each participant produced for each error. To assess the *reliability* of their responses, we use a triangulation question which asks for their *confidence* in the correctness of the produced artifacts. In addition to these artifacts, for each error, we also measure the *perceived difficulty* of each debugging task (i.e., bug diagnosis and bug fixing), the *time spent* with each debugging task, and their opinion on whether bug diagnosis or repair will ever be *fully automated* for the given error. The questions that we ask for each error are shown in figures 16 and 17.

To measure *qualitative attributes*, we utilize the 5-point Likert scale [119]. A 5-point Likert scale allows to measure otherwise qualitative properties on a symmetric scale where each item takes a value from 1 to 5 and the distance between each item is assumed to be equal.¹⁶ For instance, for Question 12 in Figure 17, we can assign the value 1 to *Not at all difficult* up to the value 5 for *Extremely difficult*. An average difficulty of 4.7 would indicate that most respondents feel that this particular error is *very* to *extremely difficult* to fix while only few think it was *not at all difficult*.

We note that all data, including time, is self-reported rather than recorded during observation. Participants fill questionnaires and provide the data on their own. This allowed us to conduct the study fully remotely without supervision while they could work in their every-day environment. Since freelancers are typically paid by the hour, Upwork provides mechanisms to ensure that working time is correctly reported. While self-reports might be subject to cognitive bias, they also reduce observer-expectancy bias and experimenter bias [144]. Perry et al. [145] conducted an observational study with 13 software developers in four software development departments and found that the time diaries which were created by the developers *correspond sufficiently* with the time diaries that were created by observers. In other words, in the software development domain self-reports correspond sufficiently with independent researcher observations.

We checked the *plausibility* of the submitted patches by executing the complete test suite and the previously failing test case. We checked the *correctness* of the submitted patches using internal code reviews. Two researchers spent about two days discussing and reviewing the patches together. Moreover, we designate a patch as incorrect only if we can provide a rationale. Generally, every qualitative analysis was conducted by at least one researcher and cross-checked by at least

¹⁶However, the Likert-scale is robust to violations of the equal distance assumption: Even with larger distortions of perceived distances between items (e.g., slightly vs. moderately familiar), Likert performs close to where intervals are perceived equal [143].

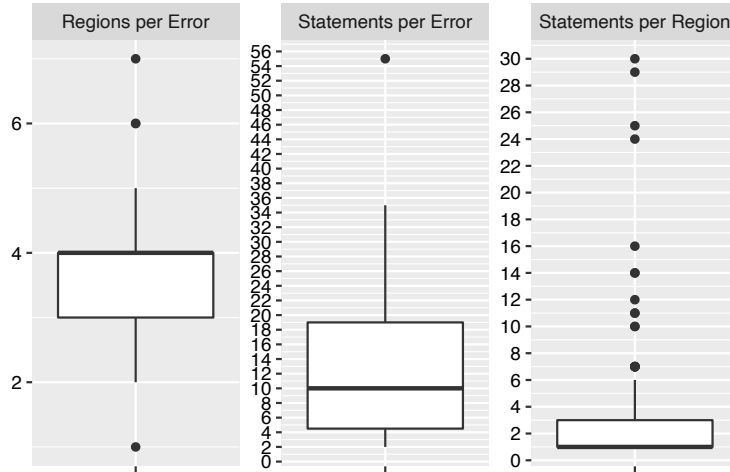


Figure 19: Boxplots showing the number of contiguous regions per bug diagnosis (left), the number of statements per diagnosis (middle), and the number of statements per contiguous region (right). For example, `file.c:12-16,20` specifies six statements in two contiguous regions.

one other researcher.

3.3.2 Study Results

Overall, *27 real errors* in 2 open-source C programs were diagnosed and patched by *12 participants* who together spent *29 working days*.

RQ1 Time and Difficulty

Our data on debugging time and difficulty can be used in cost-effective user-studies that set out to show how a novel debugging aid improves the debugging process in practice. We also elicit causes of difficulties during the manual debugging process.

Time and Difficulty. *On average, participants rated an error as moderately difficult to explain (2.8) and slightly difficult to patch (2.3). On average, participants spent 32 and 16 minutes on diagnosing and patching an error, respectively. There is a linear and positive relationship between perceived difficulty and the time spent debugging. As we can see in Figure 18, participants perceived four errors to be very difficult to diagnose. These are three functional errors and one crash. Participants spent about 55 minutes diagnosing the error that was most difficult to diagnose. However, there are also nine errors perceived to be slightly difficult to diagnose with the main cluster situated between 15 and 20 minutes of diagnosis time. Participants perceived one (functional) error as very difficult to patch and spent about 40 minutes patching it. However, there are also two bugs perceived to be not at all difficult to patch and took about five minutes.*

Bug diagnosis is twice as time-consuming as bug fixing; participants spent two-third of their time diagnosing the bug and only a third of the time to patch.

Why are some errors very difficult? There are four errors rated as *very difficult to diagnose*. In many cases, missing documentation for certain functions, flags, or data structures were mentioned as reasons for such difficulty. Other times, developers start out with an incorrect hypothesis before moving on to the correct one. For instance, the crash in `grep.3c3bdace` is caused

by a corrupted heap, but the crash location is very distant from the location where the heap is corrupted. The crash and another functional error are caused by a simple operator fault. *Three of the four bugs* which are very difficult to diagnose are actually *fixed in a single line*. For the only error that is both very difficult to diagnose *and patch*, the developer-provided patch is actually very complex, involving 80 added and 30 deleted lines of code. Only one participant provided a correct patch.

Developers perceived bug diagnosis to be difficult due to poor code documentation and incorrect (root cause) hypothesis.

RQ2 Fault Locations and Patches

Our data on those program locations which practitioners need to explain how an error comes about (i.e., fault locations) can be used for a more *realistic evaluation* of automated fault localization, and motivates the development of techniques that point out *multiple* pertinent fault locations. Our data on multiple patches for the same error can be used to evaluate auto-repair techniques, and motivates research in automated repair and its integration with automated regression test generation to circumvent the considerable human error.

Fault Locations. *The middle 50% of consolidated bug diagnoses references three to four contiguous code regions, many of which can appear in different functions or files. In other words, practitioners often reference multiple statements to explain an error. A contiguous code region is a consecutive sequence of program statements. In most cases, the regions for one error are localized in different functions and files. As shown in Figure 19, the majority of contiguous regions (below the median) contain only a single statement, the middle 50% contains between 1 and 3 statements. A typical bug diagnosis references 10 statements.*

Developers reference multiple (10) statements and multiple (three to four) code regions to explain an error.

Patches. *While 282 out of 290 (97%) of the submitted patches are plausible and pass the provided test case, only 182 patches (63%) are actually correct and pass our code review.*¹⁷ A *correct patch* does not introduce new errors and does not allow to provide other test cases that fail due to the same error. We determined *correctness* by code review and *plausibility* by executing the failing test case. For each incorrect patch, we also give a reason as to why it is incorrect and whether the test case passes. DBGBENCH provides several examples of correct and incorrect patches for an error and a high-level description of the changes done to the code. An example is shown in Figure 26-c.

Figure 20 shows that the *median proportional plausibility* is 100%, meaning that for the majority of errors (above the median), all patches that participants submit pass the provided test case. Even for the middle 50% of errors, more than 90% of patches are plausible. However, the *median proportional correctness* is 69%, meaning that for the majority of errors (above the median), only 69% of patches submitted by participants pass our code review. For the middle 50% of errors only between 45% and 82% of patches are actually correct.

¹⁷Note that participants were asked to ensure the plausibility of their submitted patch by passing the provided

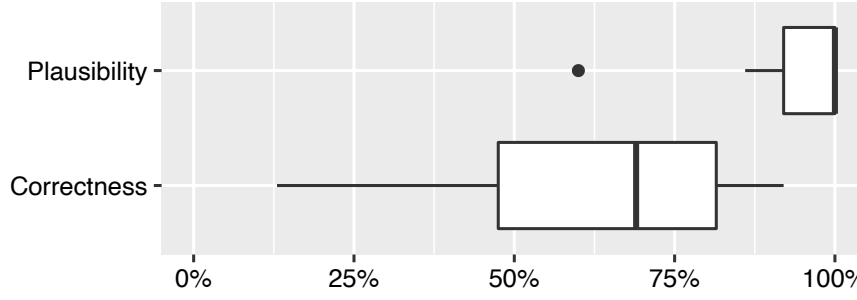


Figure 20: Boxplots showing the proportional patch correctness and plausibility. For instance, if the proportional patch correctness for an error is 50%, then half of the patches submitted for this error are correct. The boxplot shows the proportional plausibility and correctness *over all 27 errors*.

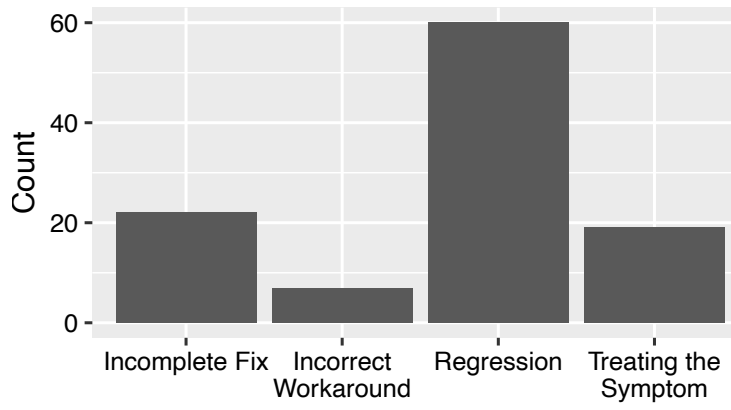


Figure 21: Histogram showing reasons why 108 of patches that were submitted by participants failed our code review.

Figure 21 shows that more than half of the incorrect patches (60 of 108) actually introduce regressions. A *regression* breaks existing functionality; we could provide a test that fails but passed before. 22 patches do not fix the error completely. An *incomplete fix* does not patch the error completely; we could provide a test that fails with and without the patch because of the bug. 19 patches are treating the symptom rather than fixing the error. A patch is *treating the symptom* if it does not address the root cause. For instance, it removes an assertion to stop it from failing. Seven (7) patches apply an incorrect workaround than fixing the error. An *incorrect workaround* changes an artifact that is not supposed to be changed, like a third-party library.

Most developer-provided patches (97%) are plausible (i.e. pass the provided failing test), but less than two-third (63%) of these patches are actually correct.

RQ3 Bug Diagnosis Strategies

For each error, we asked participants which concrete steps they took to understand the runtime actions leading to the error. We analyzed 476 different responses for this question and we observed the following bug diagnosis strategies.

test case.

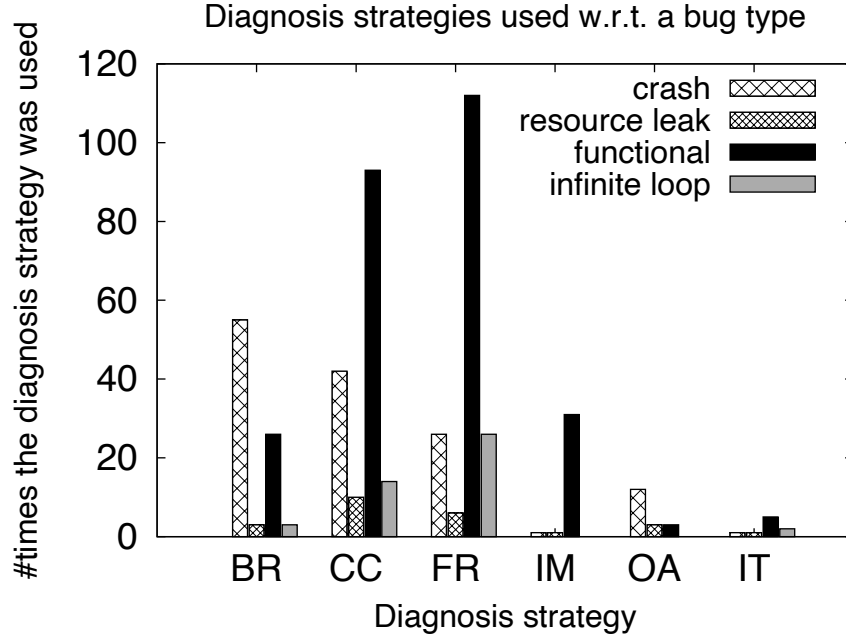


Figure 22: Diagnosis strategies for different error types.

Classification. We extend the bug diagnosis strategies that have been identified by Romero and colleagues [146, 147]:

- (FR) *Forward Reasoning*. Programmers follow each computational step in the execution of the failing test.
- (BR) *Backward Reasoning*. Programmers start from the unexpected output following backwards to the origin.
- (CC) *Code Comprehension*. Programmers read the code to understand it and build a mental representation.
- (IM) *Input Manipulation*. Programmers construct a similar test case to compare the behavior and execution.
- (OA) *Offline analysis*. Programmers analyze an error trace or a coredump (e.g. via valgrind, strace).
- (IT) *Intuition*. Developer uses her experience from a previous patch.

Specifically, we identified the Input Manipulation (IM) bug diagnosis strategy. Developers would first modify the failing test case to construct a passing one. This gives insight into the circumstances required to observe the error. Next, they would compare the program states in both executions. IM is reminiscent of classic work on automated debugging [148], which might again reflect the potential lack of knowledge about automated techniques that have been available from the research community for over a decade.

Frequency. We discovered that *forward reasoning and code comprehension (FR+CC) are the most commonly used* diagnosis strategies in our study. The number of usage of different bug diagnosis strategies is shown in Figure 22 for the different error types. We observe that *past experience (IT) is used least frequently*. Many participants used input modification (IM) as diagnosis strategy. Therefore, the integration of automated techniques that implement IM (e.g. [148]) into mainstream debugger will help improve debugger productivity.

Forward reasoning and code comprehension (FR+CC) are the most commonly used bug diagnosis strategies among developers.

Error Type. We observe that forward reasoning (FR) is the most commonly used diagnosis strategy for bugs reflecting infinite loops (26 out of a total 45 responses). Intuitively, there is no last executed statement which can be used to reason backwards from. *Out of a total 137 responses for crash-related bugs, we found that backward reasoning (BR) was used 55 times.* Intuitively, the crash location is most often a good starting point to understand how the crash came about. For functional errors, 112 responses, out of a total 270 responses, reflect forward reasoning (FR). If the symptom is an unexpected output, the actual fault location can be very far from print statement responsible for the unexpected output. It may be better to start stepping from a location where the state is not infected, yet. Finally, we observed that input modification (IM) strategy was used for 31 out of 270 scenarios to diagnose functional errors. This was to understand what distinguishes the failing from a passing execution.

The choice of diagnosis strategy depends on the error type, for instance, forward reasoning (FR) was mostly employed to diagnose functional errors and infinite loops.

Tools. Every participant used a combination of trace-based and interactive debugging. For resource leaks, participants further used tools such as *valgrind* and *strace*. We also observed that participants use bug diagnosis techniques that have been automated previously [148], albeit with manual effort, to narrow down the pertinent sequence of events.

Developers mostly used interactive or trace-based debuggers (e.g. gdb), but often used specialized tools (e.g. valgrind) for specific errors (e.g. memory leaks).

RQ4 Repair Ingredients

Out of 290 submitted patches, 100 (34%) exclusively affect the control flow, 87 (30%) exclusively affect the data flow, while the remaining 103 patches (36%) affect both, control and data flow.

Control Flow. In automated repair research, the patching of control flow is considered tractable because of the significantly reduced search space [149]: Either a set of statements is executed or not. The frequency with which participants fix the control flow may provide some insight about the effectiveness of such an approach for the errors provided with DBGBENCH. The control flow is modified by 200 patches (69%). Specifically, a branch condition is changed by 126 patches and the loop or function flow is modified by 38 patches.¹⁸ A new if-branch is added by 86 patches whereupon, in many cases, an existing statement is then moved into the new branch or a new function call is added.¹⁹

Data Flow. The data flow is modified by 187 of 290 submitted patches (64%). Specifically, 57 patches *modify* a variable value or function parameter. GENPROG [33] copies, moves, or deletes existing program statements, effectively relying on the Plastic Surgery Hypothesis (PSH) [150]. In our study, the PSH seems to hold. 44 patches *move* existing statements while 29 patches *delete* existing statements. However, 73 patches *add* new variable assignments while 40 patches add new

¹⁸Examples of changing the loop or function flow are adding a return, exit, continue, or goto statement.

¹⁹Note that one patch can modify several statements!

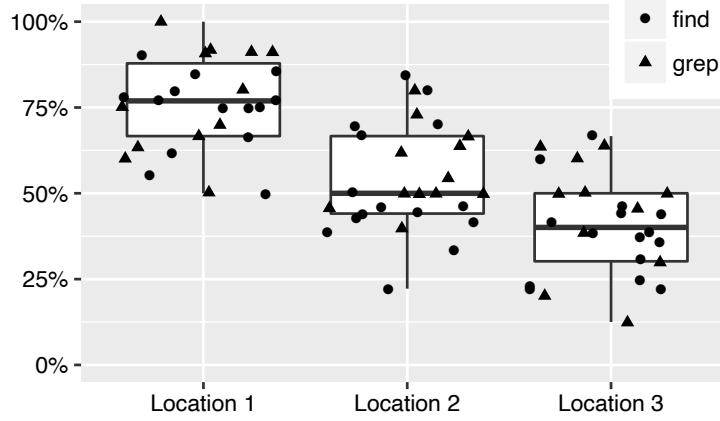


Figure 23: Proportional agreement on Top-3 most suspicious fault locations showing box plots with jitter overlay (each shape is one of 27 errors).

function calls, for instance to report an error or to release resources. A completely new variable is declared in 27 patches. Only 8 patches introduce complex functions that need to be synthesized.

A third (36%) of submitted patches affect both data and control flow, while another third exclusively affect data flow (30%) or control flow (34%), respectively.

Patch Complexity. On average, a submitted patch contained six (6) Changed Lines of Code (CLoC). The median patch contained 3 CLoC. The mean being to the far right of the median points to a skewed distribution. Indeed, many patches are not very complex but there are a few that require more than 50 CLoC.

Most bug fixes are not very complex, most patches contain three to six CLoC.

RQ5 Consensus on Debugging

We investigate whether there is consensus among the developers during fault localization, diagnosis and fixing. Suppose, there is not. Then, how should there ever be consensus on whether an automated debugging technique has produced the *correct* fault locations, the *correct* bug diagnosis, or the *correct* patch for an error?

Consensus on Fault Locations. *For most errors (above the median), more than 75% of participants independently localize the same location as pertinent to explain the error (Location 1).* For each error, we count the proportion of participants independently reporting a fault location. Sorted by proportion, we get the top-most, 2nd-most, and 3rd-most suspicious locations (Location 1–3). For the middle 50% of errors, between half and two third of participants independently report *the same* 2nd-most suspicious location while between one third and half of participants still independently report *the same* 3rd-most suspicious location. However, note that a participant might mention less or more than three locations. The consensus of professional software developers on the fault locations suggests that every bug in DBGBENCH is *correctly localized* by at least two (2) specific statements. These locations can serve as ground truth for the evaluation of automated fault localization tools.

Consensus on Bug Diagnosis. *10 of 12 participants (85%) give essentially the same*

diagnosis for an error, on average. In other words, there can be consensus on whether an automated technique has produced a correct bug diagnosis. These participants are *very confident* (3.7 on the Likert scale) about the correctness of their diagnosis. On the other hand, participants who provide a diagnosis that is different from the consensus are only *slightly confident* (2.4) about the correctness of their diagnosis. The ability to generate a consolidated, concise bug diagnosis that agrees with the majority of diagnoses as provided by professional software engineers shows that understanding and explaining an error is no subjective endeavor. The consolidated bug diagnoses in DBGBENCH can serve as the ground truth for information that is relevant to practitioners.

*There is significant consensus on fault locations (75%)
and bug diagnoses (85%) among developers.*

Consensus on Bug Fix. *For 18 of 27 bugs (67%), there is at least one other correct fix that conceptually differs from the original developer-provided patch.* In other words, often, there are several ways to patch an error correctly, syntactically and semantically. It might seem obvious that a correct patch can syntactically differ from the patch that is provided by the developer. However, we also found correct patches that conceptually differ from the original patch that was provided by the original developer. For each bug in DBGBENCH, there are 1.9 conceptually different correct fixes on average. Five (5) bugs (19%) have at least two other conceptually different but correct fixes. For instance, to patch a null pointer reference, one participant might initialize the memory while another might add a null pointer check. To patch an access out-of-bounds, one participant might double the memory that is allocated initially while others might reallocate memory only as needed. For the error in `grep.9c45c193` (Figure 26), some participants remove a negation to change the outcome of a branch while others set a flag to change the behavior of the function which influences the outcome of the branch.

*There is no consensus on bug fixing among developers;
two-third (67%) of all bug fixes are conceptually different from that of the original developer.*

RQ6 Debugging Automation

We examine if participants believe that the diagnosis and repair of these bugs can be fully automated and the reasons for their beliefs.

Automation of Bug Diagnosis. *Most professional software developers do not believe that the diagnosis of the errors in DBGBENCH can be fully automated.* The boxplot in Figure 24 provides more details. For the middle 50% of errors, one to two third of participants believe that bug diagnosis can be automated. However, this varies with bug type. For instance, for 5 of 7 crashes, more than three quarter of participants believe that the crash can be explained intuitively. Functional bugs seem much more involved and intricate such that for the median functional bug only one third of participants believe that it will ever be explained intuitively by a machine.

Automation of Program Repair. *For the median bug in DBGBENCH, only a quarter of participants believes that it can be fixed reliably by a machine* (cf. Figure 25). Again this differs

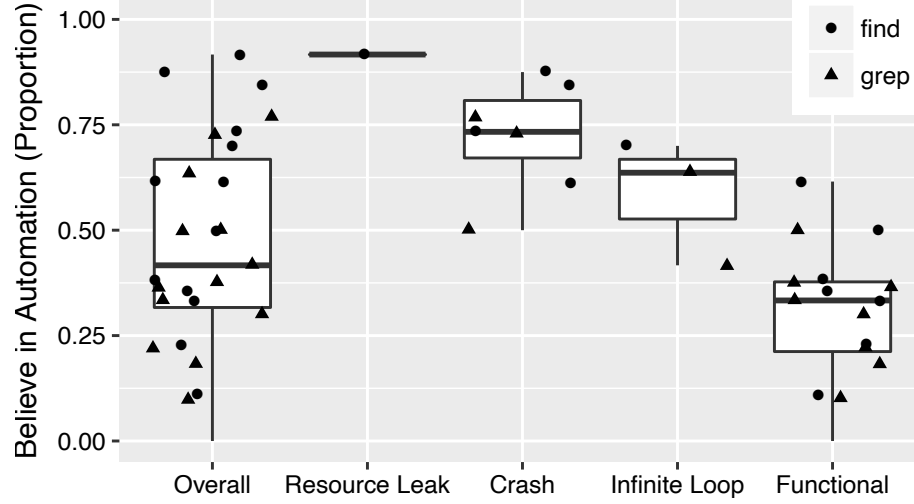


Figure 24: Distribution over all errors (of a certain type) of the proportion of participants who believe that a certain error may ever be *explained intuitively* by a machine. Each shape in the jitter plot overlay represents one error.

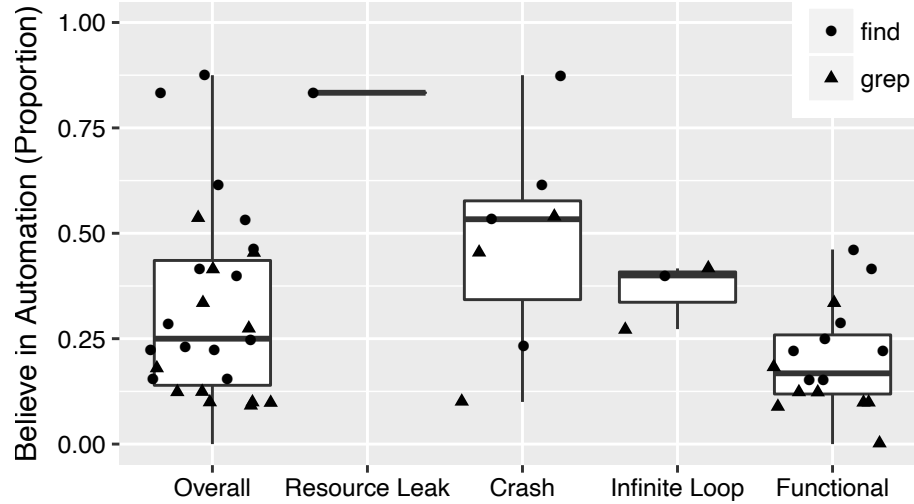


Figure 25: Distribution over all errors (of a certain type) of the proportion of participants who believe that a certain error may ever be *fixed reliably* by a machine. Each shape in the jitter plot overlay represents one error.

by bug type. While half the participants would still think that 4 of 7 crashes can be fixed reliably by a machine, functional bugs are believed to be most difficult to be fixed automatically. The single resource leak appears to be most easy to fix reliably.

For most bugs, developers believe bug diagnosis can be automated (33 to 67%) but they believe automatic bug fixing can not be reliably automated (33%).

Points in Favor of Automation. Participants believe in automation of bug diagnosis and repair primarily due to the following reasons: (i) sophisticated static or dynamic analysis, (ii) the possibility to check contracts at runtime and (iii) the possibility to cross check with a passing execution. For instance, for the *resource leak*, where most participants agree that an automated diagnosis and patch is achievable, they reflect on the possibilities of dynamic or static analysis

Bug Description: Find “-mtime [+n]” is broken (behaves as “-mtime n”)

(a) Bug Report and Test Case	(b) Bug diagnosis and Fault Locations	(c) Examples of (in)correct Patches
<p>Find “-mtime [+n]” is broken (behaves as “-mtime n”)</p> <p>Lets say we created 1 file each day in the last 3 days:</p> <pre>\$ touch tmp/a -t \$(date -date="yesterday" +"%Y/%m/%d/%H/%M") \$ touch tmp/b -t \$(date -date="2 days ago" +"%Y/%m/%d/%H/%M") \$ touch tmp/c -t \$(date -date="3 days ago" +"%Y/%m/%d/%H/%M")</pre> <p>Running a search for files younger than 2 days, we expect</p> <pre>\$./find tmp -mtime -2 tmp tmp/a</pre> <p>However, with the current <code>grep-version</code>, I get</p> <pre>\$./find tmp -mtime -2 tmp/b</pre> <p>Results are the same with <code>+n</code> or <code>n</code></p>	<p>If <code>find</code> is set to print files that are strictly younger than n days (<code>-mtime -n</code>), it will instead print files that are exactly n days old. The function <code>get_comp_type</code> actually increments the argument pointer <code>timearg</code> (<code>parser.c:3175</code>). So, when the function is called the first time (<code>parser.c:3109</code>), <code>timearg</code> still points to <code>'-</code>'. However, when it is called the second time (<code>parser.c:3038</code>), <code>timearg</code> already points to <code>'n'</code> such that it is incorrectly classified as <code>COMP_EQ</code> (<code>parser.c:3178</code>; exactly n days).</p>	<p>Example of Correct Patches</p> <ul style="list-style-type: none"> • Copy <code>timearg</code> and restore after first call to <code>get_comp_type</code>. • Pass a copy of <code>timearg</code> into first call of <code>get_comp_type</code>. • Pass a copy of <code>timearg</code> into call of <code>get_relative_timestamp</code>. • Decrement <code>timearg</code> after the first call to <code>get_comp_type</code>. <p>Example of Incorrect Patch</p> <ul style="list-style-type: none"> • Restore <code>timearg</code> only if classified as <code>COMP_LT</code> (<i>Incomplete Fix</i> because it does not solve the problem for <code>-mtime +n</code>).

Figure 26: An excerpt of DBGBENCH. For the error `find.66c536bb`, we show (a) the bug report and test case that a participant receives to reproduce the error, (b) the bug diagnosis that we consolidated from those provided by participants (including fault locations), and (c) examples of ways how participants patched the error correctly or incorrectly.

to track the lifetime of resources and to discover the right location in the code to release the resources. For the seven *crashes*, many participants believe in automation via systematic analysis that tracks changes in variable values and explains the crash through these changes. Intuitively, this captures the mechanism employed in dynamic slicing. Besides, participants think of the possibility of having contracts (e.g. a range of expected variable values) in the source code and checking their satisfiability during a buggy execution to explain the error. For *functional bugs*, the most common rationale in favor of automation was due to the possibility to compare a buggy execution with a passing execution.

Points Against Automation. *The majority of participants did not believe in automation due to the lack of a complete specification and due to the difficulty in code comprehension.* For *functional bugs*, most participants think that an automated tool cannot explain such bugs intuitively. This is because such tools are unaware of correct behaviours of the respective program. Similarly, for automated program repair, participants think that it is impossible for a tool to change or add any functionality to a buggy program. Moreover, even in the presence of a complete specification, participants do not believe in automated repair due to the challenges involved in code comprehension (e.g. the meaning of a variable or statement in the code). Finally, the difficulty to analyze side-effects of a fix is also mentioned as a hindrance for automated bug repair.

Developers do not believe in automatic bug diagnoses and repair, due to the lack of a complete specification and the difficulty in code comprehension.

3.4 A Benchmark for Debugging Tools

As our study participants agree on so many points (see **RQ5**), one can actually treat their joint diagnosis and other bug features as *ground truth*: For each bug, the joint diagnosis and fix is

Ordinal Rank	(1)	(2)	(3)	(4)	(5)	(6)
Line in parser.c	3055	3057	3058	3061	3062	3067
Ochiai Score	0.98	0.98	0.98	0.98	0.98	0.98
Ordinal Rank	(7)	(8)	(9)	(10)	(11)	(12)
Line in parser.c	3094	3100	3103	3107	3109	3112
Ochiai Score	0.98	0.98	0.98	0.98	0.98	0.98

Figure 27: Top-12 most suspicious statements in file parser.c. There are 26 statements with the same suspiciousness (0.98), including parser.c:3109 mentioned by our participants.

```

static boolean get_relative_timestamp (const char *str, ...)
3038     if (get_comp_type(&str, ...))
    ...
static boolean parse_time (...)
3099     const char *timearg = argv[*arg_ptr];
3100 +   const char *orig_timearg = timearg;
    ...
3109     if(get_comp_type(&timearg, &comp))
    ...
    ...
3126 +   timearg = orig_timearg;
3127     if (!get_relative_timestamp(timearg, ...))
3128         return false;

```

Figure 28: Original developer-patch for bug in Figure 26.

what a debugging tool should aim to support and produce. To support *realistic evaluation* of automated debugging and repair approaches, we have compiled a benchmark named DBGBENCH, which encompasses the totality of the data collected from the debugging sessions of developers. An excerpt of DBGBENCH for a specific bug is shown in Figure 26. Using the example in Figure 26, we illustrate how DBGBENCH (and thus the results of our study) can be used to evaluate fault localization and automated repair tools.

Evaluating Automated Fault Localization: Statistical fault localization techniques produce a list of statements ranked according to their suspiciousness score. We used *Ochiai* [151] to compute the suspiciousness score for the statements in the motivating example (Figure 27). In order to evaluate the effectiveness of a fault localization (FL) technique, researchers first need to identify the *statements which are actually faulty* and determine the rank of the highest ranked faulty statement (i.e., wasted effort [39]).

Using DBGBENCH, researchers can use the *actual* fault locations that practitioners point out. For instance, in Figure 26-b, the highest ranked *faulty* statement in DBGBENCH is parser.c:3109. As it turns out, this statement is also within the set of most suspicious statements with an Ochiai-score of 0.98. Thus, DBGBENCH provides a useful artifact to validate the effectiveness of FL techniques.

Without DBGBENCH, the “faulty” statements are typically identified as those statements which were changed in the original patch to fix error [39, 14, 110]. However, this assumption may not always hold [15]. Figure 28 shows the original patch for our example. *No statement* was changed in the buggy version. The original patch merely introduced new statements. In fact, only 200 of 290 patches (69%) submitted by our participants modify at least one statement that is referenced in the consolidated bug diagnosis.

Evaluating Automated Program Repair: Automated Program Repair (APR) techniques automatically generate a patch for a buggy program such that all test cases pass. We ran RELIFIX

[152] to generate the following patch for our motivating example. The generated patch directly uses the original value assigned to `timearg` before calling `get_comp_type` the second time. This is in contrast to the developer-provided patch in Figure 28 which copies `timearg` and restores it after the first call to `get_comp_type`. Is the auto-generated patch correct?

```
static boolean parse_time (...)
3127 - if (!get_relative_timestamp(timearg, ...))
3127 + if (!get_relative_timestamp(argv[*arg_ptr], ...))
3128     return false;
```

Using DBGBENCH, we can evaluate the *correctness* of the auto-generated patch. First, for many participant-provided patches we provide new test cases that fail for an incorrect patch even if the original test cases all pass. For instance, we constructed a new test case for the incorrect-patch-example in Figure 26-c, where the actual output of `./find -mtime +n` is compared to the expected output. Executing it on RELIFIX’s auto-generated patch above, we would see it passes. We can say that the auto-generated patch does not make the same mistake. While we can still not fully ascertain the correctness of the patch, we can at least be confident the auto-repair tool does not make the same mistakes as our participants.

However, to establish patch *correctness* with much more certainty and to understand whether practitioners would actually *accept* an auto-generated patch, we suggest to conduct a user study. Within such a user study, DBGBENCH can significantly reduce the time and effort involved in the manual code review since the available bug diagnosis, simplified and extended regression test cases, the bug report, the bug diagnosis, fault locations, and developer-provided patches are easily available. For instance, while RELIFIX’s auto-generated patch conceptually differs from the original in Figure 28, it is easy to determine from the provided material that the auto-generated patch is in fact correct.

DBGBENCH includes time taken by professional software engineers to fix real-world software bugs. We can use these timing information to evaluate the *usefulness* of an automated program repair tool. To this end, we can design an experiment involving several software professionals and measure the reduction in debugging time while using an automated program repair tool.

DBGBENCH Artifact: DBGBENCH was given the *highest badge* by the ESEC/FSE Artifact Evaluation Committee. DBGBENCH is the first human-generated benchmark for the qualitative evaluation of automated fault localization, bug diagnosis, and repair techniques.

Objectives. The objectives of the DBGBENCH artifact are two-fold:

1. To facilitate the sound replication of our study for other researchers, participants, subjects, and languages, we publish our battle-tested, formal experiment procedure, and effective strategies to mitigate the common pitfalls of such user studies. To the same effect, we publish tutorial material (videos and slides), virtual infrastructure (Docker), and example questionnaires which elicit the studied debugging artifacts.
2. To facilitate the effective evaluation of automated fault localization, diagnosis, and repair techniques with respect to the judgement of human experts, we publish all collected data. This includes the fault locations, bug diagnoses, and patches that were provided by the

practitioners. For each error, it also includes the error-introducing commit, the original and a simplified bug report, a test case failing because of the error, and the developer-provided patch. Moreover, this artifact contains our reconciled bug diagnoses, our classification of the patches as correct and incorrect together with the general fixing strategies, and which rationale we have to classify a patch as incorrect.

Provided data. Specifically, we make the following data available:

- The *benchmark summary* containing the complete list of errors, their average debugging time, difficulty, and patch correctness, human-generated explanations of the runtime actions leading to the error, and examples of correct and incorrect fixes, sorted according to average debugging time.
- The *complete raw data* containing the responses to the questions in Figure 16 and Figure 17 for each debugging session:
 - the error ID to identify error,
 - the participant ID to identify participant,
 - the timestamp to follow participants accross errors,
 - fault Locations, bug diagnosis, and patches,
 - the confidence in the correctness of their diagnosis or patch,
 - the difficulty to diagnose or patch the error,
 - the time taken to diagnose or patch the error,
 - what would have helped reducing diagnosis or patch time,
 - the steps taken to arrive at the diagnosis or patch,
 - the tools used to arrive at the diagnosis or patch,
 - the problems if they could not diagnose or patch the error,
 - whether he/she believes that generating the diagnosis or patch for this error would ever be automated,
 - why he/she believes so,
 - the code familiarity as it increases over time,
 - the diagnosis techniques used, and
 - how he/she ensured the correctness of the submitted patch,
- The *complete cleaned data* containing for each error:
 - the regression-introducing commit,
 - the simplified and original bug reports,
 - the important fault locations,
 - the reconciled bug diagnosis,

- the original, developer-provided patch,
 - the patches submitted by participants,
 - our classification of patches as correct or incorrect,
 - our rationale of classifying a patch as incorrect, and
 - test cases that expose an incorrect patch.
- An *example questionnaire* to elicit the raw data above.
 - The *Docker virtual infrastructure* and instructions how to use it.
 - The *tutorial material*, incl. slides, videos, and readme files.

Disclaimer DBGBENCH is a first milestone towards the *realistic evaluation* of tools in software engineering that is grounded in practice. DBGBENCH can thus be used as necessary reality check for in-depth studies. When conducting user studies, DBGBENCH can significantly reduce the time and cost that is inherent in large user studies involving professional software engineers. In the absence of user studies, DBGBENCH allows in-depth evaluations while minimizing the number of potentially unrealistic assumptions. For example, we found a high consensus among participants on the location of the faults. They would often point out several contiguous code regions rather than a single statement. We also found no overlap of fault locations with fix locations. Existing error-benchmarks might assume that an artificially injected fault (i.e., a single mutated statement) or fix locations (i.e., those statements changed in the original patch) are representative of realistic fault locations. DBGBENCH dispenses with such assumptions and directly provides those fault locations that practitioners would point out with high consensus. DBGBENCH allows the realistic evaluation of automated debugging techniques that is grounded in practice.

However, we would strongly suggest to also utilize other benchmarks, such as COREBENCH [14] or Defects4J [110], for the *empirical evaluation*. Only an empirical evaluation, the use of a sufficiently large representative set of subjects, allows to make empirical claims about the efficacy of an automated debugging technique. Going forward, we hope that more researchers will produce similar realistic benchmarks which take the practitioner into account. To this end, we also publish our battle-tested, formal experiment procedure, effective strategies to mitigate common pitfalls, and all our material. We believe that no single research team can realistically produce a benchmark that is both representative and reflects the realities of debugging in practice. Hence, we propose that similar benchmarks be constructed alongside with *user studies*. Software engineering research, including debugging research, must serve the needs of users and developers. User studies are essential to properly evaluate techniques that are supposed to automate tasks that are otherwise manual and executed by a software professional. Constructing new benchmarks as artifacts during user studies would allow to build an empirical body of knowledge and at the same time minimize the cost and effort involved in user studies.

3.5 Limitations and Threats to Validity

Generalizability of Findings. For the results of our experiment we do not claim generalizability of the findings. We decided on two subjects to limit the time a participant spends in our study

to a maximum of three working days and to help participants to get accustomed to at most two code bases. We chose 27 reproducible real errors in single-threaded open-source C projects where bug reports and test cases are available. Our findings may not apply to (irreproducible) faults in very large, distributed, multi-threaded or interactive programs, to short-lived errors that do not reach the code repository, to errors in programs written in other languages, or to errors in programs developed within a software company. We see DBGBENCH as an intermediate goal for the community rather than the final benchmark.

Hence, we encourage fellow researchers to extend and conduct similar experiments in order to build an empirical body of knowledge and to establish the means of evaluating automated debugging techniques more faithfully, without having to resort to unrealistic assumptions [11] during the evaluation of an automated technique. To facilitate replication, the questionnaires, the virtual infrastructure, and the tutorial material are made available [116]. While we do not claim generality, we do claim that DBGBENCH is the first dataset for the evaluation of automated diagnosis and repair techniques with respect to the judgment of twelve expert developers. In the future, DBGBENCH may serve as subject for *in-depth experiments*.

In empirical research, in-depth experiments may mistakenly be taken to provide little insight for the academic community. However, there is evidence to the contrary. Beveridge observed that “more discoveries have arisen from intense observation than from statistics applied to large groups” [153]. This does not mean that research focusing on large samples is not important. On the contrary, both types of research are essential [154].

Cognitive Bias. Results may suffer from cognitive bias since participants fill a questionnaire for each errors, such that all responses are *self-reported* [155, 156, 118]. However, Perry et al. [145] found that self-reports produced by the developers, in their case, often corresponded sufficiently with observations independently recorded by a researcher. As standard mitigation of cognitive bias,

1. we *avoid leading questions* and *control for learning effects*,
2. we *reinforce confidentiality* for more truthful responses,
3. we *use triangulation* [126] by checking the consistency of replies to separate questions studying the same subject,
4. we mostly *utilize open-ended questions* that provide enough space for participants to expand on their replies
5. and otherwise *utilize standard measures* of qualitative attributes, such as the Likert scale [119].

While our experiment was fairly long-running, we also suggest to replicate our study at a different point in time with different participants to check whether the responses are *consistent*.

Observer-Expectancy Bias. To control for expectancy bias where participants might behave differently during observation, we conducted the study remotely in a virtual environment with minimal intrusion. Participants were encouraged to use their own tools. We also emphasized that there was no “right and wrong behavior”.

Imposed Time Bound. We suggested the participants to complete an error in 45 minutes so as to remain within a 20 hours time frame. Some errors would take much more time. So, given more time, the participants might form a better understanding of the runtime actions leading to the error and produce a larger percentage of correct patches. However, participants told us that they felt comfortable to diagnose and patch each error within the stipulated time-bound. They tended to stretch the bounds whenever necessary, taking time from errors that were quickly diagnosed and patched.

3.6 Related Work

There are several articles exploring the opinions of practitioners on research in automated software engineering. However, to the best of our knowledge none explores the rationale *why* practitioners do not wish to adopt automated debugging techniques, which capabilities an automated bug diagnosis tool should have beyond simple fault localization, or which qualities an auto-generated patch must have so that developers would actually accept it.

In 2015, Perscheid et al. [19] set out to determine whether the state-of-practice has since improved, and conducted a survey involving 303 practitioners *to study time spent and tools* used in debugging practice. The middle fifty percent of their respondents spend 20% to 40% of their work time for debugging while the next largest group reported 40% to 60%. Like our study, most practitioners reported to only use trace-based (e.g., `printf`) or interactive debugging (e.g., `gdb`) regularly. The authors also find that most respondents have *never* used any automated technique such as fault localization, slicing, or likely invariants. However, Perscheid et al. note that the sample might not be representative to allow a general conclusion whence (as is customary in the empirical sciences) we replicate and extend their study as part of our larger study.

Begel and Zimmermann [124] conducted a *prospective study* that investigates potential software engineering research problems that practitioners find interesting. The authors published 145 questions ranked in the order of importance that professional software engineers would like to ask a data scientist. These questions were mostly about software users, engineers, processes, and practices. For instance, the question perceived as most important was: “How do users typically use my application?” Lo et al. [123] conducted a *retrospective study* that explores how developers rate the relevance of 517 actual software engineering research papers. The authors elicit reasons as to why developers consider certain research as “unwise”: Researchers might overestimate the relevance of a problem, or fail to consider the cost, side effects, or actionability of a solution. Thus, *it is important to explore practitioners’ expectations and beliefs before attempting to automate a manual activity.*

The bulk of *automated debugging research* is on Automated Fault Localization (AFL), that is, techniques that produce a ranked list of suspicious locations in the program. A recent survey cites more than 400 publications on AFL [39]. This is in stark contrast to the finding of Perscheid et al. [19] that most practitioners have *never* used an AFL tool. Parnin and Orso [112] shed light on this dichotomy and conducted a small user study with an AFL tool. Even when the fault was ranked artificially high, participants did not find the tool more effective than traditional debugging. After inspecting *the* fault in the list of suspicious locations, 9 of 10 participants would still spend another 10 minutes before stopping to provide a diagnosis. The authors found that

beyond AFL, developers wanted the relevant context and variable values. More recently, Kochhar et al. [41] investigated the necessary capabilities of an AFL tool such that practitioners would consider its adoption. Three quarter of surveyed practitioners would investigate no more than the Top-5 ranked statements — which should contain the faulty statement at least three out of four times and take less than one minute to compute.²⁰ Most practitioners find it *crucial* that the AFL tool explains the rationale *why* the user should consider a suspicious location.

Previous work established that practitioners are interested in automated diagnosis tools with capabilities that go beyond simple AFL [112, 41]. In this work, we follow up and explore *which* capabilities both, automated bug diagnosis *and repair* tools should have. In contrast to previous work, we want to shed light on Lieberman’s debugging scandal without limiting ourselves to AFL. We find out exactly *what* practitioners want that would improve their debugging experience. We find that most practitioners do not believe that debugging will ever be fully automated and provide their rationales.

In 2015, Perscheid et al. [19] set out to determine whether the state-of-the-practice had since improved and could only answer in the negative: Debugging remains what it was 50 years ago, largely manual, time-consuming, and a matter of trial-and-error rather than automated, efficient, and systematic. We believe that, at least in part, the debugging scandal is brought about by the absence of user studies and the *assumptions* that researchers had to make when evaluating the output of a machine without involving the human. Only recently, several studies have uncovered that many of these assumptions are not based upon empirical foundations [11, 12, 13, 14, 15]. In this work, we attempt to remedy this very problem.

While several researchers investigated *debugging strategies* that developers generally employ in practice, it remains unclear whether developers who independently debug the same error essentially localize the same faulty statements, provide the same explanation (i.e., diagnosis) and generate the same patch. Perscheid et al. [19] visited four companies in Germany and conducted think-aloud experiments with a total of eight developers during their normal work. While none was formally trained in debugging, all participants used a simplified, implicit form of *Scientific Debugging* [157]: They mentally formulated hypotheses and performed simple experiments to verify them. Katz and Anderson [147] classify bug diagnosis strategies broadly into *forward reasoning* where the programmer forms an understanding of what the code should do compared to what it actually does, and *backward reasoning* where the programmer reasons backwards starting from the location where the bug is observed (e.g., [158]). Romero et al. [146] explore the impact of graphical literacy on the choice of strategy. Lawrance et al. [26] model debugging as a predator following scent to find prey. The authors argue that a theory of navigation holds more practical value for tool builders than theories that rely on mental constructs. Gilmore et al. [159] studied different models of debugging and their assumptions. The authors argue that the success of experts at debugging is not attributed to better debugging skills, but to better comprehension. Our evaluation of the impact of “Code Comprehension” in this study further validates these findings.

Several colleagues have investigated debugging as a human activity. Ko et al. [160] observed how 10 developers understand and debug unfamiliar code. The authors found that developers

²⁰Supposing a tool must have a minimum satisfaction rate of 80%.

interleaved three activities while debugging, namely, code search, dependency analysis and relevant information gathering. Layman et al. [161] investigated how developers use information and tools to debug, by interviewing 15 professional software engineers at Microsoft. The authors found that the interaction of hypothesis instrumentation and software environment is a source of difficulty when debugging. Parnin and Orso [112] also conducted an empirical study to investigate how developers use and benefit from automated debugging tools. The authors found that several assumptions made by automated debugging techniques do not hold in practice. While these papers provide insights on debugging as a human activity, none of these studies provides the data and methods that would allow researchers to evaluate debugging tools against fault locations, bug diagnosis, and patches provided by actual software engineering professionals.

Many researchers have proposed numerous tools to support developers when debugging, but only a few have evaluated these tools with user studies, using real bugs and professional developers. For instance, between 1981 and 2010, Parnin and Orso [112] identified only a handful of articles [162, 163, 164, 165, 166, 167] that presented the results of a user study: Unlike this study, none of these studies involved actual practitioners and real errors. In our own literature survey, studying whether the state-of-the-research has since improved, we could identify only *three (3) papers* that conduct user studies with actual practitioners and real errors in the last five years.²¹ Two articles employed user studies to evaluate the acceptability of the auto-generated patches [114, 113] while one had practitioners to evaluate the effectiveness of an auto-generated bug diagnosis [115].²² We believe that this general absence of user studies is symptomatic of the difficulty, expense, and time spent conducting user studies. This is the problem we address with DBGBENCH.

3.7 Discussions and Future Work

In this chapter, we provide empirical evidence on program debugging in software practice, our evaluation results overwhelmingly reveal that *debugging in practice is (still) a largely manual and time-consuming process*. Developers spend half of their working time debugging source code which they are often not very familiar with. After successfully reproducing a reported bug, developers spend most of their time trying to understand the chain of runtime actions leading to bug. Many practitioners find their own style of debugging to be trial-and-error rather than systematic. Twenty years after “The Debugging Scandal” [24], debugging in practice is still what it was five decades ago.

The state-of-the-practice is in stark contrast to the many tools and techniques that have been developed in several decades of software engineering research on automated debugging. For bug diagnosis, we find that many capabilities that practitioners expect of an automated tool have been addressed and well-researched by the software engineering research community. For instance, most practitioners would like a tool that can localize the fault while automated fault localization

²¹We surveyed the following conference proceedings: ACM International Conference on Software Engineering (ICSE’11–16), ACM SIGSOFT Foundations of Software Engineering (FSE’11–16), ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA’11–16), IEEE/ACM International Conference on Automated Software Engineering (ASE’11–16), and International Conference on Software Testing, Verification and Validation (ICST’11–16) and identified 82 papers on automated debugging or software repair only 11 of which conducted user studies. From these only three (3) involved software engineering professionals *and* real errors.

²²Tao et al. [115] measured debugging time and the percentage of correct repairs provided by the participants, after they had received an auto-generated patch as bug diagnosis

is also the most popular and a long-standing topic in research of automated debugging [39]. Then, *why is there little or no adoption of automated debugging and repair tools?*

The empirical evidence from our study suggests that *most practitioners reject the notion that bug diagnosis or repair will ever be fully automated*. The most cited reason is that some errors are difficult to understand even by a human, specifically when the system is too complex and the machine cannot access all relevant information or when the correct behavior is unspecified and the machine cannot distinguish bug from feature. For instance, if your mobile’s GPS is off by 5km and after debugging it is off by 5m, has the bug really been fixed?

Building on these empirical evidence, we see five clear directions to address the concerns of practitioners: 1) *to assume full automation only for certain types of bugs*, 2) *to allow more interaction with the developer*, 3) *to provide some proof of correctness for the provided diagnosis or patch*, 4) *to develop techniques that account for the presence of uncertainty*, and 5) *to introduce and improve systematic training in debugging*.

Do not Assume Full Automation: Our first recommendation is that researchers should *assume full automation only for certain types of bugs*. To this end, researchers could develop an ontology for prevalent classes of software errors and an automated scheme that classifies an error as easy or difficult to diagnose or patch with existing debugging tools.

Practitioners accept that certain bugs can be fully automatically diagnosed or fixed. For instance, if the program crashes because an array is accessed out of bounds, developers would allow a tool to automatically introduce the required bounds check. Syntactic errors might be auto-corrected similar to spelling errors on the phone. For other (difficult) errors, more interaction or some proof of correctness is needed.

Despite their wariness for full automation, practitioners also provide plenty of reasons why at least some debugging tasks might be fully automated at some point in the future. Apart from advances in artificial intelligence, practitioners believe that not every bug is new and common mistakes are repeated, which allows machines to *learn from history*. Practitioners also believe that *program analysis* will improve and allow to mine more relevant information from the faulty program.

Allow for More Interaction: As our second recommendation, we state that researchers should *investigate how automated debugging tools can interact with the user to access information that is otherwise inaccessible, such as the developer’s experience with similar bugs, or which program behavior would be expected*. Practitioners would like interactive debugging assistants that render the developer more efficient in solving a bug, instead of debugging automatons that seek to substitute the developer, or assistants that slow down the developer with too many false positives or too much irrelevant information. The most prominent work in this direction is Ko and Meyer’s WHYLINE [163]. Beyond WHYLINE, such interaction could enable the assistant to learn from common bugs and mistakes in earlier debugging sessions.

To support practitioners further, future research thus could investigate intuitive visualizations of relevant information that can be easily navigated; researchers might want to develop techniques that produce a natural language explanation (e.g. in English) of pertinent program information to improve the automation of bug diagnosis; researchers could investigate how a tool can analytically determine the expected value of a variable (some information should be present in the test suite,

other could be requested from the user); and researchers should develop techniques that can determine the most general environment or conditions under which the error can be reproduced.

Context, Rationales, and Correctness: Our third recommendation is that researchers should *develop techniques that explain the process which leads an automated debugging or repair tool to the specific diagnosis or patch that is provided*. They also should *integrate automated repair and regression testing techniques to provide some proof that the patch did not introduce new errors* (e.g., [168, 169]). In general, practitioners would like some evidence that the produced diagnosis or patch is correct, including simple, yet comprehensive reports, that explain why the error occurs, that are easy to navigate and to understand.

A deeper understanding of the process that underlies the automated debugging tool and of its “rationale” would build trust in the correctness of the produced output. Otherwise, practitioners even explicitly ask for auto-generated test cases that exercise the patch and pass after the patch, in order to convince the user that the auto-generated patch does not introduce any new bugs.

In terms of automated program repair, a common problem statement in research is to produce a patch that passes the failing test suite. While the auto-generated patch is plausible, it might not be correct [138]. However, practitioners are interested in properties of auto-generated patches beyond patch correctness. In fact, a property that was mentioned twice as often as correctness was patch readability: *An auto-generated patch should appeal to the human, be well-documented, well-structured, easy to understand, and minimal*. As mentioned before, practitioners want some evidence that the patch really fixes the bug and that it does not introduce new bugs. A bug fixing tool should document how it fixes the bug, the potential impact of the patch, and relevant context information.

To further improve automated repair techniques, researchers could *investigate a wider variety of properties of auto-generated diagnoses and patches to increase adoption in the industry*. For instance, researchers could leverage search algorithms from multi-objective search-based software engineering (e.g., [170]) to produce patches that are not only correct (i.e., pass all test cases) but also are readable, minimal, and adhere to the existing coding standards. Beyond fault localization and repair, an automated bug diagnosis tool should be able to distinguish actual from expected values, find out under which most general circumstances the bug can be reproduced, and determine the side-effects of an error.

Account for Uncertainty: In our study, developers highlight the frequent absence of specifications and complexity of the system to be debugged as reasons why they do not believe in automated debugging. This is in contrast to research of automated debugging, where we often assume 1) access to the complete state of a buggy system and 2) that the test suite or specification is complete with respect to the buggy behavior. In practice, neither assumption may ever hold.

As our fourth recommendation, we thus state that *researchers should investigate automated debugging in the presence of uncertainty*. This is along the lines of the recent work by Elbaum and Rosenblum, who called for testing techniques that can work in the presence of uncertainty [136]. Our study shows that we may just as well extend this call to the debugging community [21].

Teach Debugging: Finally, our study shows that much more training for systematic debugging is required: *Educators and trainers should teach systematic methods for debugging*. Our study reaffirms that one third of software development time is spent on debugging; yet, the amount of

space spent on debugging in courses or textbooks is a much lesser fraction, if not zero. Developers should be formally trained in systematic debugging processes [157] as well as the principles of the most important debugging and analysis techniques, such that they can deploy them as needed. This calls for better abstractions and descriptions of debugging processes, as well as evaluating these with students and practitioners.

Note, though, that better debugging is not just a matter of productivity, it is also a matter of *risk*. In our study, 30% of respondents admit to “trial and error” debugging. This effectively means that they neither understand how the code works nor how the bug came to be. If 30% of programmers were to confess that they program by trial and error—imagine the uproar this would cause. Yet, in the end, debugging a program always means to (re)write a program—and thus, trial-and-error debugging is the same as trial-and-error programming, with all the risks that follow.

In summary, this study suggests that debugging in practice is still in a deplorable state. Given how much time practitioners spend on debugging, it is surprising how little we know about debugging [38]. This study shows that the realities of debugging and the needs of practitioners are far more complex and fine-grained than what a convenient abstraction (or benchmark) would cover. With our study, we hope to have shed some light into this under-researched area. This study provides some insights on what practitioners need and what they believe, we encourage researchers to investigate and fulfill these needs, in order to challenge these beliefs. With our DBGBENCH benchmark, we hope to contribute some ground truth to guide the development and evaluation of future debugging tools towards the needs and strategies of professional developers. We encourage researchers to challenge and refine our findings, and possibly repeat and extend this study as desired. The DBGBENCH benchmark as well as all other study data is available at our project site:

<https://dbgbench.github.io>

Locating Faults with Program Slicing: An Empirical Analysis

This chapter is taken, directly or with minor modifications, from our 2021 EMSE paper *Locating Faults with Program Slicing: An Empirical Analysis* [171]. My contribution in this work is as follows: (I) original idea; (II) partial implementation; (III) evaluation.

“Synergy: The combined effect of individuals in collaboration that exceeds the sum of their individual effects.”

— Stephen Covey

4.1 Introduction

In the past 20 years, the field of *automated fault localization* (AFL) has found considerable interest among researchers in Software Engineering. Given a program failure, the aim of fault localization is to suggest locations in the program code where a fault in the code causes the failure at hand. Locating a fault is an obvious prerequisite for removing and fixing it; and thus, *automated* fault localization brings the promise of supporting programmers during arduous debugging tasks. Fault localization is also an important prerequisite for *automated program repair*, where the identified fault locations serve as candidates for applying the computer-generated patches [33, 35, 55, 56].

The large majority of recent publications on automated fault localization fall into the category of *statistical debugging* (also called *spectrum-based fault localization* (SBFL)), an approach pioneered 15 years ago [27, 54, 53]. A recent survey [172] lists more than 100 publications on statistical debugging in the past 15 years. The core idea of statistical debugging is to take a set of passing and failing runs, and to record the program lines which are executed (“covered”) in these runs. The stronger the correlation between the execution of a line and failure (say, because the line is executed only in failing runs, and never in passing runs), the more we consider the line as “suspicious”.

As an example, let us have a look at the function `middle`, used in [27] to introduce the technique (see Figure 29). The `middle` function computes the middle of three numbers `x`, `y`, `z`; Figure 29 shows its source code as well as statement coverage for few sample inputs. On most

	■: covered statements	x	3	1	3	5	5	2	
1	int middle(x, y, z) {	y	3	2	2	5	3	1	
2	int x, y, z;	z	5	3	1	5	4	3	
3	int m = z;		■	■	■	■	■	■	3
4	if (y < z) {		■	■	■	■	■	■	4
5	if (x < y)		■	■	□	□	■	■	5
6	m = y;		□	■	□	□	□	□	6
7	else if (x < z)		■	□	□	□	■	■	7
8	m = y;		■	□	□	□	□	■	8
9	} else {		□	□	■	■	□	□	9
10	if (x > y)		□	□	■	■	□	□	10
11	m = y;		□	□	■	□	□	□	11
12	else if (x > z)		□	□	□	■	□	□	12
13	m = x;		□	□	□	□	□	□	13
14	}		□	□	□	□	□	□	14
15	return m;		■	■	■	■	■	■	15
16	}		✓	✓	✓	✓	✓	✗	

Figure 29: Statistical debugging illustrated [46]: The `middle` function takes three values and returns that value which is greater than or equals the smallest and less than or equals the biggest value; however, on the input (2, 1, 3), it returns 1 rather than 2. Statistical debugging reports the faulty Line 8 (in **bold red**) as the most suspicious one, since the correlation of its execution with failure is the strongest.

inputs, `middle` works as advertised; but when fed with $x = 2$, $y = 1$, and $z = 3$, it returns 1 rather than the middle value 2. Note that the statement in Line 8 is incorrect and should read $m = x$. Given the runs and the lines covered in them, statistical debugging assigns a *suspiciousness score* to each program statement—a function on the number of times it is (not) executed by passing and failing test cases. The precise function it uses differs for each statistical debugging technique. Since the statement in Line 8 is executed most often by the failing test case and least often by any passing test case, it is reported as most suspicious fault location.

Statistical debugging, however, is not the first technique to automate fault localization. In his seminal paper titled “Programmers use slices when debugging” [42], Mark Weiser introduced the concept of a *program slice* composed of data and control dependencies in the program. Weiser argued that during debugging, programmers would start from the location where the error is observed, and then proceed backwards along these dependencies to find the fault. In a debugging setting, programmers would follow *dynamic* dependencies to find those lines that actually impact the location of interest in the *failing run*. In our example (Figure 30), they could simply follow the dynamic dependency of Line 15 where the value of `m` is unexpected, and immediately reach the faulty assignment in Line 8. Consequently, on the *example originally introduced to show the effectiveness of statistical debugging* (Figure 29), the older technique of dynamic slicing is just as effective (see Figure 30).

Thus, in this chapter, we investigate the fault localization effectiveness of *the most effective* statistical debugging formulas against dynamic program slicing (both introduced in Chapter 2). A few researchers have empirically evaluated the fault localization effectiveness of different slicing algorithms [173, 174]. However, they did not compare the effectiveness of slicing to that of statistical debugging. To the best of our knowledge, this is the *first empirical study* to evaluate

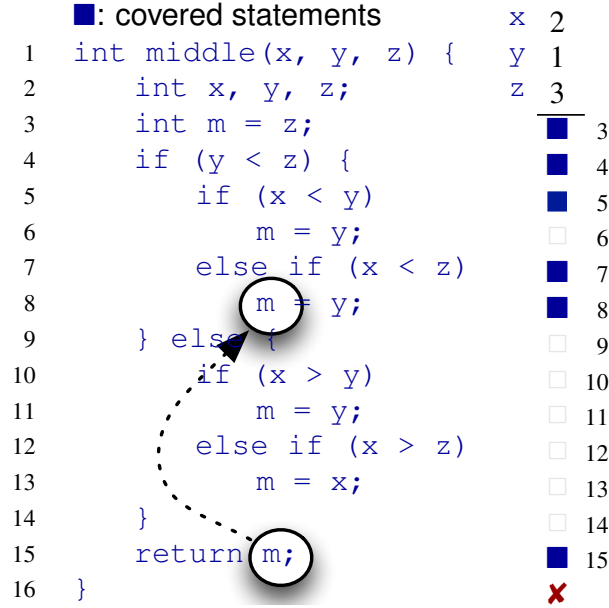


Figure 30: Dynamic slicing illustrated [46]: The `middle` return value in Line 15 can stem from any of the assignments to `m`, but only those in Lines 3 and 8 are executed in the failing run. Following back the dynamic dependency immediately gets the programmer to Line 8, the faulty one.

the fault localization effectiveness of program slicing versus (one of) the most effective statistical debugging formulas. This is also one of the *largest empirical studies* of fault localization techniques, evaluating hundreds of faults (706) in C programs.

To ascertain the fault localization effectiveness of these AFL techniques (program slicing and statistical debugging), firstly, we conducted an evaluation using 457 bugs from four benchmarks. We collect empirical evidence from the performance of both techniques using real world programs and faults. Secondly, using the evidence from this evaluation, we proposed a *hybrid strategy* where programmers first examine the top (five) most suspicious locations from statistical debugging, and then switch to dynamic slices. Next, we evaluate the effectiveness of the hybrid approach, as well as the influence of other factors such as error type and the number of faults on the fault localization effectiveness of all three techniques. In our evaluation, we use four benchmarks with 35 tools, 46 programs and 457 bugs to compare fault localization techniques against each other. This set of bugs comprises of 295 real single faults, 74 injected single faults, and 88 injected multiple faults containing about four faults per program, on average. In total, we had 709 program faults. Our takeaway findings are as follows:

1. **Top ranked locations in statistical debugging can pinpoint the fault.** If one is only interested in a *small set of candidate locations*, statistical debugging frequently pinpoints the faults, it correctly localizes 33% of faults after inspecting *only the single most suspicious code location*. It outperforms dynamic slicing in the top 5% of the most suspicious locations, by localizing faults in *twice as many* buggy programs as dynamic slicing. In our experiments, looking at only the top 5% of the most suspicious code locations, statistical debugging would reveal faults for 6% of all buggy programs, twice as many as slicing (3% of buggy programs). This result is important for *automatic program repair* (APR) techniques, as the search for possible repairs can only consider a limited set of candidate locations; also, the

repair attempt is not necessarily expected to succeed.

2. **If one *must* fix a (single-fault) bug, dynamic slicing is more effective.**²³ In our experiments, dynamic slicing is 62% more likely to find the fault location earlier than statistical debugging, for single faults. In absolute terms, locating faults along dynamic dependencies requires programmers to examine on average 21% of the code (40 LoC); whereas the most effective statistical debugging techniques require 26% (51 LoC). Not only is the average better; the effectiveness of dynamic slicing also has a much lower standard deviation and thus is more predictable. Both features are important for *human debuggers*, as they eventually must find and fix the fault: If they follow the dynamic slice from the failing output, they will find the fault quicker than if they examine locations whose execution correlates with failure. Moreover, dynamic slicing needs only the failing run, whereas statistical debugging additionally requires multiple similar passing runs. Although dynamic slicing is more effective on single faults, statistical debugging performs better on multiple faults (see **RQ7**).
3. **Programmers can start with statistical debugging, but should quickly switch to dynamic slicing after a few locations.** In our experiments, it is a *hybrid* strategy that works best: First consider the top locations of statistical debugging (if applicable), and then proceed along the dynamic slice. In our experiments, *the hybrid approach is significantly more effective than both slicing and statistical debugging*. For most errors (98%), the hybrid approach localizes the fault within the top-20 most suspicious statements, in contrast, both slicing and statistical debugging will localize faults for most errors (98%) after inspecting about five times as many statements (100 LoC). Notably, the hybrid approach is more effective than statistical debugging and dynamic slicing, regardless of the error type (real/artificial) and the number of faults (single/multiple) in a buggy program, (see **RQ6** and **RQ7**).

The remainder of this chapter is organized as follows:

1. Section 4.2 presents an approach that merges both dynamic slicing and statistical debugging into a *hybrid strategy*, where the developer switches to slicing after investigating a handful of the most suspicious statements reported by statistical debugging.
2. We describe our evaluation setup (see Section 4.3) and empirically evaluate the fault localization effectiveness of dynamic slicing, statistical debugging and our hybrid approach (**RQ1** to **RQ5** in Section 4.4).
3. We conduct an empirical study on the effect of error type and the number of faults on the effectiveness of AFL techniques. We examine the difference between evaluating an AFL technique on *real vs. artificial faults* (**RQ6** in Section 4.4), as well as *single vs. multiple faults* (**RQ7** in Section 4.4).

²³In our evaluation, dynamic slicing is more effective than SBFL for single faults. However, other factors such as multiple faults (see **RQ7**), test generation [175], test reduction [176] and program sizes may influence its effectiveness (see Section 4.5).

In Section 4.5, we discuss the limitations and threats to the validity of this work. Section 4.6 presents closely related work. Finally, Section 4.7 closes with consequences and future research directions.

The contributions and findings of this chapter are important for debugging and program repair stakeholders. Programmers, debugging tools and automated program repair (APR) tools need *effective* fault localization techniques, in order to reduce the amount of time and effort spent (automatically) diagnosing and fixing errors. In particular, these findings enable APR tools, debuggers and programmers to be effective and efficient in bug diagnosis and bug fixing.

4.2 A Hybrid Approach

Even though dynamic slicing is generally more effective than statistical debugging, we observe that statistical debugging can be highly effective for some bugs, especially when inspecting only the top most suspicious statements. For instance, statistical debugging can pinpoint a single faulty statement as the most suspicious statement for about 40% of the errors in **IntroClass** and **SIR**, i.e. a developer can find a faulty statement after inspecting only one suspicious statement (see Figure 36). This is further illustrated by the clustering of some points in the rightmost corner below the diagonal line of the comparison charts (see Figure 35).

In this work, we assume that a programmer in the end *has* to fix a bug, and a viable “alternative” method is following the dependencies by (dynamic) slicing. To this end, we investigate a *hybrid* fault localization approach which leverages the strengths of both dynamic slicing and statistical debugging. The goal is to improve on the effectiveness of both approaches by harnessing the power of statistical correlation and dynamic program analysis. The hybrid approach first reports the *top most suspicious statements* (e.g. top five statements) before it reports the statements in the dynamic slice computed w.r.t. the symptomatic statement.

The concept of examining only the top most suspicious statements is also backed by user studies on statistical fault localization. In a recent survey [177], Kochhar et al. found that three quarter of surveyed practitioners would investigate *no more than the top-5 ranked statements*—which should contain the faulty statement at least three out of four times—before switching to alternative methods. This is also confirmed by the study of Parnin and Orso [178], who observed that programmers tend to transition to traditional debugging (i.e., finding those statements that impact the value of the symptomatic statement) after failing to locate the fault within the first N top-ranked most suspicious statements. This transition is exactly what the hybrid approach provides.

Specifically, the hybrid approach proceeds in two phases. In the first phase, it reports the top N (e.g. $N = 5$) most suspicious statements, obtained from the ordinal ranking²⁴ of a statistical fault localization technique. Then, if the fault is not found, it proceeds to the second phase where it reports the symptom’s dynamic backward dependencies. In the second phase, we only report statements that have not already been reported in the first phase.²⁵

Weakness of Statistical Debugging: The hybrid approach is capable of overcoming the weaknesses of statistical debugging. Statistical fault localization techniques are sensitive to the

²⁴In ordinal ranking, lines with the same score are ranked by line number.

²⁵This is to avoid duplication of inspected statements, i.e. avoid double inspection.

■: covered statements		x	3	2			
1	int middle(x, y, z) {	y	3	1			
2	int x, y, z;	z	5	3	Tarantula	Ochiai	Naish2
3	int m = z;	■	■		0.500	0.707	0.500
4	if (y < z) {	■	■		0.500	0.707	0.500
5	if (x < y)	■	■		0.500	0.707	0.500
6	m = y;	□	□		0.000	0.000	0.000
7	else if (x < z)	■	■		0.500	0.707	0.500
8	m = y;	■	■		0.500	0.707	0.500
9	} else {	□	□		0.000	0.000	0.000
10	if (x > y)	□	□		0.000	0.000	0.000
11	m = y;	□	□		0.000	0.000	0.000
12	else if (x > z)	□	□		0.000	0.000	0.000
13	m = x;	□	□		0.000	0.000	0.000
14	}	□	□		0.000	0.000	0.000
15	return m;	■	■		0.500	0.707	0.500
16	}	✓	✗				

Figure 31: Test suite sensitivity of statistical debugging. Let us consider the `middle` function with fault in line 8 (in **bold red**), given a small test suite containing two test cases (3, 3, 5) and (2, 1, 3). Then, statistical debugging reports *all* executed lines 3, 4, 5, 7, 8, and 15 as the “most” suspicious statements, since they are all strongly correlated to the failure.

size and variance of the accompanying test suites [179]. Statistical debugging is less efficient when the accompanying test suite is small or achieves low or similar coverage. To reduce the time wasted in search of the fault in these cases, the hybrid approach reports only the Top- N most suspicious statements, then proceeds to dynamic slicing. For instance, Figure 31 depicts for our motivating example how the effectiveness of statistical fault localization depends on the provided test suite. Given one passing and one failing test case, statistical debugging correlates all six executed statements for the failing test case (2, 3, 4, 7, 8, 15), as the *most* suspicious ranked statements. Conservatively, the programmer needs to inspect half the program statements before finding the fault. Although a large test suite and high coverage is desirable for statistical fault localization, for real programs this is not always available. Often only one or two failing test cases are actually available [14].

Meanwhile, the hybrid approach with the same test suite improves the programmer’s effectiveness. Assuming $N = 2$, the programmer inspects the first two statements before following the dependency from the symptomatic statement in Line 15. She finds the fault after inspecting only three statements. In contrast, using statistical debugging she would find the fault after investigating five statements (using ordinal ranking).

Weakness of Dynamic Slicing: Program slices can become very large [45]. Generally, programmers using dynamic slicing become ineffective when the fault is located relatively far away from the symptomatic statement. However, our proposed hybrid approach can overcome this limitation by leveraging statistical debugging which can point to any statement in the program however far from the symptomatic statement. This improves the chances of finding the fault quickly by first applying statistical correlation before dynamic analysis.

Figure 32 illustrates this weakness. This modified variant of the `middle` function contains another fault (in Line 5) which is exposed by the same test suite. Since the return value for the failing test case (1,2,3) is unexpected, we mark Line 15 as the slicing criterion. In this example, the operator fault is located relatively far from the slicing criterion. The programmer following backward dependencies from the symptom has to inspect three statements (lines 8, 7, and 5) in

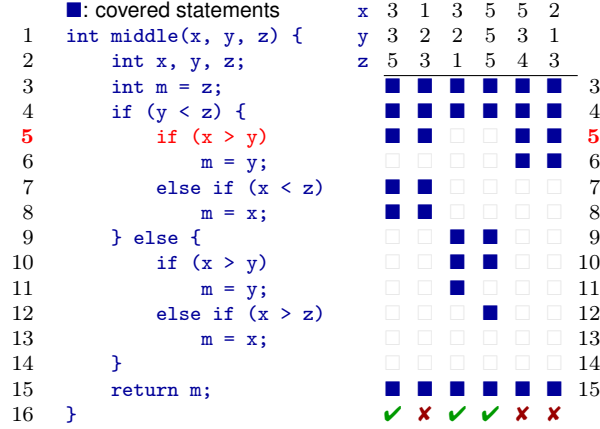


Figure 32: Weakness of dynamic slicing: long dependence chain between fault and symptom. This is a variant of the faulty `middle` function with an operator fault in Line 5 (in **bold red**). Following back the dynamic dependency gets the programmer to Line 5 (the fault) after inspecting 2–3 statements—(5,6) or (5,7,8) depending on the failing test case.

addition to the slicing criterion. On the other hand, our hybrid approach with $N \geq 1$ has to inspect only a single statement before the fault is located.

4.3 Evaluation Setup

Let us evaluate the effectiveness of all three fault localization techniques and the influence of the number of faults and error type on the effectiveness of AFL techniques. Specifically, we ask the following research questions:

RQ1 Effectiveness of Dynamic Slicing: How effective is dynamic slicing in fault localization, i.e. localizing fault locations in buggy programs?

RQ2 Effectiveness of Statistical Debugging: Which statistical formula is the most effective at fault localization?

RQ3 Comparing Statistical Debugging and Dynamic Slicing: How effective is the most effective statistical formula in comparison to dynamic slicing?

RQ4 Sensitiveness of the Hybrid Approach : How many suspicious statements (reported by statistical debugging, i.e. Kulczynski2) should a tool or developer inspect before switching to slicing?

RQ5 Effectiveness of the Hybrid Approach: Which technique is the most effective in fault localization? Which technique is more likely to find fault locations earlier?

RQ6 Real Errors vs. Artificial Errors: Does the type of error influence the effectiveness of AFL techniques? Is there a difference between evaluating an AFL technique on real or artificial errors?

RQ7 Single Fault vs Multiple Faults: What is the effect of the number of faults on the effectiveness of AFL techniques? Is there a difference between evaluating an AFL technique on single or multiple fault(s)?

In this chapter, we evaluate the performance of statistical debugging, dynamic slicing and the hybrid approach in the framework of Steimann, Frenkel, and Abreu [180] where we fix the granularity of fault localization at *statement level* and the fault localization mode at *one-at-a-time* (except for multiple faults in **RQ7**). In this setting with real errors and real test suites, the provided test suites *may not be coverage adequate*, e.g. they may not execute all program statements. Fault localization effectiveness is evaluated as *relative wasted effort* based on the ranking of units in the order they are suggested to be examined (see Section 4.3.4 for more details).

4.3.1 Implementation

Let us provide implementation details for each AFL technique.

Dynamic Slicing Implementation: The approximate dynamic slice is computed using **Frama-C**,²⁶ **gcov**, **git-diff**, **gdb**, and several Python libraries. Given the preprocessed source files of a C program, **Frama-C** computes the static slices for each function and their call graphs as DOT files. The **gcov**-tool determines the executed/covered statements in the program. The **git-diff**-tool determines the changed statements in the patch and thus the faulty statements in the program. The **gdb**-tool allows to derive coverage information even for crashing inputs and to determine the slicing criterion as the last executed statement. Our Python script intersects the statements in the static slice and the set of executed statements to derive the approximate dynamic slice. We use the Python libraries **pygraphviz**²⁷, **networkx**,²⁸ and **matplotlib**²⁹ to process the DOT files and compute the *score* for the approximate dynamic slice.

Statistical Debugging Implementation: The statistical debugging tool was implemented using two bash scripts with several standard command line tools, notably **gcov**,³⁰ **git-diff**³¹ and **gdb**³². The differencing tool **git-diff** identifies those lines in the buggy program that were changed in the patch. If the patch only added statements, we cannot determine a corresponding faulty line. Some errors were thus excluded from the evaluation. The code coverage tool, **gcov** identifies those lines in the buggy program that are covered by an executed test case. When the program crashes, **gcov** does not emit any coverage information. If the crash is *not* caused by an infinite loop, it is sufficient to run the program under test in **gdb** and force-call the **gcov**-function from **gdb** to write the coverage information once the segmentation fault is triggered. This was automated as well. However, for some cases, no coverage information could be generated due to infinite recursion. **Gcov** also gives the number of *executable* statements in the buggy program (i.e., $|P|$).³³ Finally, our Python implementation of the scores is used to compute the fault localization effectiveness.

Hybrid Approach Implementation: The hybrid approach is implemented simply as a combination of both tools. If the top- N most suspicious statements do not contain the fault, the

²⁶<http://frama-c.com/>

²⁷<https://pygraphviz.github.io/>

²⁸<https://networkx.github.io/>

²⁹<http://matplotlib.org/>

³⁰<https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>

³¹<https://git-scm.com/docs/git-diff>

³²<https://www.gnu.org/software/gdb/documentation/>

³³The executable statements refers to statements for which coverage information are obtainable by **Gcov**, in particular, all program statements except spaces, blanks and comments.

dynamic slicing component is informed about the set of statements already inspected in the first phase. Given the unranked suspiciousness score of every executable statement in the program, the hybrid fault localizer performs an ordinal ranking of all statements. It then determines the proportion of the top N rank of suspicious statements, based on the N value of the hybrid approach. For instance, an hybrid approach with $N = 5$ takes the five topmost suspicious statements. Then, it determines the highest ranked faulty statement in the rank of all suspicious statements. If the faulty statement is in the top N suspicious positions (e.g. third position), then it reports the number of statements in the top ranked positions up till the faulty statement, as a proportion of all executable program statements.

In the case that the suspicious statement is not in the top N suspicious positions (e.g. seventh position), then it proceeds to slicing and reports the cardinality of the set union of all N top ranked statements and the number of inspected statements in the slice before the first faulty statement.

4.3.2 Metrics and Measures

Odds Ratio ψ . To establish the superiority of one technique A over another technique B , it is common to measure the effect size of A w.r.t. B . A standard measure of effect size and widely used is the *odds ratio* [181]. It “is a measure of how many times greater the odds are that a member of a certain population will fall into a certain category than the odds are that a member of another population will fall into that category” [181]. In our case, let “ A is *successful*” mean that fault localization technique A is more effective than fault localization technique B and let a be the number of successes for A , b the number of successes for B , and $n = a + b$ the total number of successes. Then, the odds ratio ψ is calculated as

$$\psi = \left(\frac{a + \rho}{n + \rho - a} \right) / \left(\frac{b + \rho}{n + \rho - b} \right)$$

where ρ is an arbitrary positive constant (e.g., $\rho = 0.5$) used to avoid problems with zero successes. There is no difference between the two algorithms when $\psi = 1$, while $\psi > 1$ indicates that technique A has higher chances of success. For example, an odds ratio of five means that fault localization technique A is five times more likely to be successful (i.e., more effective as compared to B) at fault localization than B .

The Mann-Whitney U-test is used to show whether there is a statistical difference between two techniques [182]. In general, it is a non-parametric test of the null hypothesis that two samples come from the same population against an alternative hypothesis, especially that a particular population tends to have larger values than the other. Unlike the t -test it does not require the assumption that the data is normally distributed. More specifically, it shows whether the difference in performance of two techniques is actually statistically significant.

A *cumulative frequency curve* is a running total of frequencies. We use such curves to show the percentage of errors that require examining up to a certain number of program locations. The number of code locations examined is plotted on a log-scale because the difference between examining 5 to 10 locations is more important than difference between examining 1005 to 1010 locations.

Table 1: Details of Subject Programs

Benchmark (Error Type)	Tool (Program)	Avg. Size (LoC)	#Errors	#Failing Tests	#Passing Tests
SIR (Artificial)	tcas	65.1	37	1356	58140
	print_tokens	199	3	184	12206
	print_tokens2	199.5	8	2031	30889
	tot_info	125	18	1528	17408
	schedule	160.5	4	690	9910
	schedule2	139.2	4	116	10724
IntroClass (Real; Students)	checksum	11.3	3	7	41
	digits	17.4	3	16	32
	grade	16.1	8	30	114
	median	13.5	2	8	18
	smallest	13.2	2	16	16
	syllables	11.6	2	12	20
Codeflaws (Real; Competitions)	WTLW (71A)	10.3	11	60	61
	HQ9+ (133A)	10.9	18	270	1260
	AG (144A)	24.5	13	302	202
	IB (478A)	8.6	20	31	329
	TN (535A)	61.2	9	118	778
	Exam (534A)	17.5	12	108	68
	Holidays (670A)	12.8	9	662	1118
	DC (495A)	14.5	13	96	279
	VBt(336A)	13.6	14	108	309
	PP(509B)	21.2	16	84	98
	DHMF (515B)	29.5	15	127	707
	HVW2 (143A)	17.5	16	124	707
	Ball Game (46A)	10	8	114	148
	WE (31A)	14.4	14	187	200
	LM (146B)	29.5	11	116	355
	SG (570B)	7.5	11	69	531
	WD (168A)	7.7	9	132	254
	Football (417C)	13.2	13	64	352
	MS (218A)	16.5	10	156	156
	Joysticks (651A)	12.6	8	66	246
CoREBench (Real; Developers)	core. (cut)	306	4	4	6
	core. (rm)	110	1	1	63
	core. (ls)	1605.5	2	2	73
	core. (du)	315	1	1	28
	core. (seq)	219.7	3	3	5
	core. (expr)	321	1	1	1
	core. (copy)	897	1	1	59
	find (parser)	119.3	3	3	286
	find (ftsfind)	211.5	2	2	183
	find (pred)	825	2	2	235
	grep (dfasearch)	181.5	2	2	46
	grep (savedir)	64	1	1	15
	grep (kwsearch)	77	2	2	46
	grep (main)	853.5	2	2	45
Total	35 (46)		369	9012	148767

4.3.3 Objects of Empirical Analysis

Programs and Bugs: We evaluated each fault localization technique using 45 C programs containing hundreds of (369) errors and thousands of (9012) failing tests (*cf. Table 1*). These programs were collected from four benchmarks, in particular, three benchmarks containing real world errors, namely IntroClass, Codeflaws and CoREBench, and one benchmark with artificial faults, namely the Software-artifact Infrastructure Repository (SIR). We selected these benchmarks of C programs to obtain a large variety of bugs and programs. Each benchmark contains a unique set of programs containing errors introduced from different sources such as developers, students, programming competitions and fault seeding (e.g. via code mutation). These large set of bugs allows us to rigorously evaluate each fault localization technique. The following briefly describes each benchmark used in our evaluation:

1. *Software-artifact Infrastructure Repository (SIR)* is a repository designed for the evaluation of program analysis and software testing techniques using controlled experimentation [183]. It contains small C programs, with seeded errors and test suites containing thousands of failing tests. In particular, this benchmark allows for the controlled evaluation of the effects of large test suites on debugging activities.
2. *IntroClass* is a collection of small programs written by undergraduate students in a programming course [184]. It contains six C programs, each with tens of instructor-written test suites. This benchmark allows for the evaluation of factors that affect debugging in a development scenario, especially for novice developers.
3. *Codeflaws* is a collection of programs from online programming competitions held on Codeforces.³⁴ These programs were collected for the comprehensive evaluation of debugging tools using different types of errors. It contains 3902 errors classified across 40 defect classes in total [185]. In particular, this benchmark allows for the evaluation of fault localization techniques on different defect types.
4. *CoREBench* is a collection of 70 real errors that were systematically extracted from the repositories and bug reports of four open-source software projects: Make, Grep, Findutils, and Coreutils [14].³⁵ These projects are well-tested, well-maintained, and widely-deployed open source programs for which the complete version history and all bug reports can be publicly accessed. All projects come with an extensive test suite. CoREBench allows for the evaluation of fault localization techniques on real world errors (unintentionally) introduced by developers.

Table 1 lists all the programs and bugs investigated in our study. We use six programs each from the SIR and IntroClass benchmarks. This includes `tcas` – this program is *the* most well-studied subject according to a recent survey on fault localization [172]. We selected 20 programming competitions from Codeflaws, including popular and difficult contests, such as “Tavas and Nafas (535A)” and “Lucky Mask (146B)”. From CoREBench, we used three projects,

³⁴<https://codeforces.com/>

³⁵<http://www.comp.nus.edu.sg/~release/corebench/>

Table 2: Details of Multiple Faults

Benchmark (Error Type)	Tool	# Buggy Programs	#Faults	#Failing Tests	#Passing Tests
SIR (Mutated)	tcas	37	144	19973	39523
	print_tokens	3	11	12074	316
	print_tokens2	8	28	27630	5290
	tot_info	18	64	16667	2269
	schedule	4	17	9673	927
	schedule2	4	16	8616	2224
IntroClass (Mutated)	checksum	1	4	15	1
	digits	2	7	30	2
	grade	5	22	67	23
	median	2	8	13	13
	smallest	1	4	8	8
	syllables	3	13	34	14
Total		88	338	94800	50610

namely the **Coreutils**, **Grep** and **Find** project. Notably, all projects in CoREBench come from the GNU open source C programs, in particular, these three projects contain a total of 103 tools. Due to code modularity, the program size for a single tool (e.g. `cut` in **coreutils**) contain a few hundred LoC (about 306 LoC), however, the entire code base for CoREBench is fairly large. For instance, **Coreutils**, **Grep** and **Find** have 83k, 18k and 11k LoC, respectively [14]. For each benchmark, we exempted programs where **Frama-C** *could not* construct the Program Dependence Graph (PDG). For instance, because it cannot handle some recursive or variadic method calls. In addition, we excluded an error if no coverage information could be generated (e.g., infinite loops) or the faulty statement could not be identified (e.g., *omission faults* where the patch only added statements).

Single Faults: For our evaluation (all **RQs** except **RQ7**), we used buggy programs collected from benchmarks where each program contained only a single fault, for most programs. To determine single faults in our bug dataset, for each program, we executed all tests available for a project on the fixed version of the program, in order to determine if there are any failing test cases that are unrelated to the bug at hand. Our evaluation revealed that our dataset contained mostly single bugs (368/369=99.7%). Almost all buggy program versions had exactly one fault, except for a single program – **Codeflaws** version DC 495A. For all benchmarks, only this program contained multiple faults, i.e. more than one fault. This distribution of single faults portrays the high prevalence of single faults and single-fault fixes in the wild [186].

Multiple Faults: To evaluate the effectiveness of all three fault localization techniques on multiple faults (see **RQ7**), we automatically curated a set of multiple faults using *mutation-based fault injection*, in line with the evaluation of multiple bugs in previous works [187, 188, 189, 190, 191].³⁶ We automatically mutated the original passing version of each program until we have a buggy version containing between three to five faults. In particular, we performed logical and arithmetic operator mutation on each passing version of the programs contained in the **SIR** and **IntroClass** benchmarks. Table 2 provides details of the buggy programs with multiple faults, the number of faults, as well as the number of failing and passing test cases. For each fault

³⁶To the best of our knowledge, there is no known benchmark of real-world multiple bugs.

contained in the resulting program, we store the failing test case(s) that expose the bug, as well as the corresponding patches for each fault and all faults. In total, we collected 88 programs with multiple faults containing 338 injected faults, in total. Each program in this dataset contained about four unique faults, on average. Specifically, we collected 74 and 14 programs from the **SIR** and **IntroClass** benchmarks, and injected a total of 280 and 58 faults in each benchmark, respectively. The programs with multiple faults are called **SIR-MULT** and **IntroClass-MULT**, respectively.

Minimal Patches: The user-generated patches are used to identify those statements in the buggy version that are marked faulty. In fact, Renieris and Reiss [192] recommend identifying as faulty statements those that need to be changed to derive the (correct) program that does not contain the error. For each error, only patched statements are considered faulty. All bugs in our corpus are patched with at least one statement changed in the buggy program, all *omission bugs* are exempted. Omission bugs require special handling since they are quite difficult to curate, localize and fix. Collecting patches and fault locations for omission bugs is difficult because their patches are similar to the implementation of new features. Thus, a faulty code location is unclear for these bugs in the (failing commit), this makes them even more difficult to evaluate for typical AFL techniques, including statistical debugging and dynamic slicing [193].

Slicing Criterion: All aspects of dynamic slicing can be fully automated. To this end, as the slicing criterion we chose the last statement that is executed or the return statement of the last function that is executed. For instance, when the program crashes because an array is accessed out of bounds, the location of the array access is chosen as the slicing criterion. In our implementation, the slicing criterion is automatically selected by a bash script running **gdb**.

Passing and Failing Test Cases: All programs in our dataset come with an extensive test suite which checks corner cases and that previously fixed errors do not re-emerge. For statistical debugging, we execute each of these (passing) test cases individually to collect coverage information. For dynamic slicing, we perform slicing for each failing test case.

In summary, for our automated evaluation, we used 457 errors in dozens of programs from four well-known benchmarks (see Table 1 and Table 2). Our corpus contained 46 different programs in 35 software tools. Each faulty program in our corpus had about 11 bugs, 257 failing test cases and thousands (4250) of passing test cases, on average. For single faults, we have 295 real faults and 74 injected faults. Meanwhile, we have 88 buggy programs for multiple faults, each containing about four faults, on average.

4.3.4 Measure of Localization Effectiveness

We measure *fault localization effectiveness* as the proportion of statements that do *not* need to be examined until finding the first fault. This allows us to assign a score of 0 for the worst performance (i.e., all statements must be examined) and 1 for the best. More specifically, we measure the $score = 1 - p$ where p is the proportion of statements that needs to be examined before the first faulty statement is found. Not all failures are caused by a single faulty statement. In a study of Böhme and Roychoudhury, only about 10% of failures were caused by a single statement, while there is a long tail of failures that are substantially more complex [14]. Focusing on the first faulty statement found, the *score* measures the effort to find a good starting point

to initiate the bug-fixing process rather than to provide the complete set of code that must be modified, deleted, or added to fix the failure. [172] motivates this measure of effectiveness and presents an overview of other measures.

General Measures

Ranking. All three fault localization techniques presented in this chapter produce a ranking. The developer starts examining the highest ranked statement and goes down the list until reaching the first faulty statement. To generate the ranking for *statistical debugging*, we list all statements in the order of their suspiciousness (as determined by the technique), most suspicious first. To generate the ranking for *approximate dynamic slicing*, given the statement c where the failure is observed, we rank first those statements in the slice that can be reached from c along one backward dependency edge. Then, we rank those statements that can be reached from c along two backward dependency edges, and so on. Generally, for all techniques, the *score* is computed as

$$score = 1 - \frac{|S|}{|P|}$$

where S are all statements with the same rank or less as the highest ranked faulty statement and P is the set of all statements in the program. So, S represents the statements a developer needs to examine until finding the first faulty one.³⁷

Multiple Statements, Same Rank. In most cases there are several statements that have the same rank as the faulty statement. For all our evaluations, we employ ordinal ranking, in order to effectively determine the top N most suspicious statements for each technique. This is necessary to evaluate the fault localization effectiveness of each technique, if a developer is only willing to inspect N most suspicious statements [177]. In ordinal ranking, lines with the same score are re-ranked by line numbers.³⁸ This is in agreement with evaluations of fault localization techniques in previous work [172, 194, 177].

Multiple Faults, Expense Score. For *multiple faults*, we measure fault localization effectiveness using the *expense score* [176]. The expense score is the percentage of the program (statements) that must be examined to find the *first fault*, in particular, the *first faulty statement in the first localized fault* using the ranking given by the fault localization technique. It is similar to the score employed for single faults, and it has been employed in previous evaluations of multiple faults, such as [191, 190]. Formally:

$$expense\ score = \frac{|S|}{|P|} * 100$$

where S are all statements with the same rank or less as the highest ranked faulty statement for *the first fault found* and P is the set of all executable statements in the program. So, S represents the statements a developer needs to examine until finding *the first faulty statement, for the first localized fault*. The assumption is that it is the first fault that the developer would begin fixing, thus, finding the first statement suffices for the diagnosis of all faults [176]. In our evaluation of multiple faults, the fault localization effectiveness *score* is computed similarly to single faults as

$$score_{mult} = 1 - (expense\ score/100)$$

³⁷Note that all executable program statements are ranked in the suspiciousness rank, executable statements that are not contained in the dynamic slice are ranked lowest.

³⁸Ranking ties are broken in ascending order, i.e. if both lines 10 and 50 have the same score, then line number 10 is ranked above line number 50.

Dynamic Slicing Effectiveness: We define the effectiveness of *approximate dynamic slicing*, the $score_{ads}$ according to Renieris and Reiss [192] as follows. Given a failing test case t , the symptomatic statement c , let P be the set of all statements in the program, let ζ be the approximate dynamic slice computed w.r.t. c for t , let k_{min} be the minimal number of backward dependency edges between c and any faulty statement in ζ , and let $DS_*(c, t)$ be the set of statements in ζ that are reachable from c along at most k_{min} backward dependency edges. Then,

$$score_{ads} = 1 - \frac{|DS_*(c, t)|}{|P|}$$

Algorithmically, the $score_{ads}$ is computed by (i) measuring the minimum distance k_{min} from the statement c where the failure is observed to any faulty statement along the backward dependency edges in the slice, (ii) marking all statements in the slice that are at distance k_{min} or less from c , and (iii) measuring the proportion of marked statements in the slice. This measures the part of code a developer investigates who follows backward dependencies of executed statements from the program location where the failure is observed towards the root cause of the failure.

In the approximate dynamic slice in our motivating example (Figure 2), we have $score_{ads} = 1 - \frac{1}{12} = 0.92$. The slicing criterion is $c = s_{15}$. The program size is $|P| = 12$. The faulty statement s_8 is ranked first. Statements s_7 and s_2 are both ranked third according to *modified competition ranking*³⁹. Statements s_5 and s_4 are ranked fourth and fifth, respectively, while the remaining, not executed (but executable) statements are ranked 12th.

Statistical Debugging Effectiveness: We define the effectiveness of a *statistical fault localization* technique, the $score_{sfl}$ as follows. Given the ordinal ranking of program statements in program P for test suite T according to their suspiciousness as determined by the statistical fault localization method, let r_f be the rank of the highest ranked faulty statement and P is the set of all statements in the program. Then,

$$score_{sfl} = 1 - \frac{r_f}{|P|}$$

Note that $score_{sfl} = 1 - EXAM\text{-}score$ where the well-known *EXAM-score* [46, 179] gives the proportion of statements that need to be examined until the first fault is found. Intuitively, the $score_{sfl}$ is its complement assigning 0 to the worst possible ranking where the developer needs to examine all statements before finding a faulty one.

For instance, $score_{sfl} = 1 - \frac{1}{12} = 0.92$ for our motivating example and all considered statistical debugging techniques. All statistical debugging techniques identify the faulty statement in Line 8 as most suspicious. So, there is only one top-ranked statement (Rank 1). But there are six statements with the lowest rank (Rank 12). If the fault was among one of these statements, the programmer might need to look at all statements of our small program `middle` before localizing the fault.

Hybrid Approach Effectiveness: We define the effectiveness of the hybrid approach, the $score_{hyb}$ as follows. Let R be the set of faulty statements, H be the N most suspicious statements – sorted first by suspiciousness score and then by line numbers and P is the set of all statements

³⁹In this case, when several statements have the same rank as the faulty statement, we made the conservative assumption that a developer finds the faulty statement among other statements with the same rank.

Table 3: Effectiveness of Dynamic Slicing on Single Faults

Benchmark	Score	% Errors Localized if developer inspects N most suspicious LoC			
		5	10	20	30
IntroClass	0.83	70.00	100	100	100
Codeflaws	0.78	75.30	92.71	98.79	100
CoREBench	0.85	18.52	18.52	29.63	40.74
Real	0.79	69.73	86.39	92.52	94.56
Artificial (SIR)	0.79	32.43	44.59	55.41	60.81
Avg. (Bugs)	0.774	62.23	77.99	85.05	87.77

in the program. Given the failing test case t and a statement c that is marked as symptomatic, we have

$$score_{hyb} = \begin{cases} \min(score_{sf}, N) & \text{if } R \cap H \neq \emptyset \\ 1 - |H \cup DS_*(c, t)| / |P| & \text{otherwise} \end{cases}$$

Essentially, $score_{hyb}$ computes the score for the statistical fault localization technique if the faulty statement is within the first N most suspicious statements, and the score for approximate dynamic slicing while accounting for the statements already reported in the first phase. For instance, for $N = 2$ we have $score_{hyb} = 1 - \frac{1}{12} = 0.92$ for the motivating example in Figure 29 since the fault is amongst the N most suspicious statements.

Infrastructure: We performed the experiments on a virtual machine (VM) running Arch Linux. The VM was running on a Dell Precision 7510 with a 2.7GHz Intel Core i7-6820hq CPU and 32GB of main memory.

4.4 Experimental Results

Let us discuss the results of our evaluation and their implications. All research questions (**RQs**) are evaluated using single faults, except for **RQ7** which is also evaluated on multiple faults.

RQ1 Effectiveness of Dynamic Slicing

How effective is dynamic slicing in fault localization? To investigate the fault localization effectiveness of dynamic slicing, we examined the proportion of statements a developer would *not need to* inspect after locating the faulty statement (*score* in Table 3). Then, we examine the percentage of errors for which a developer can effectively locate the faulty statement, if she inspects only N most suspicious statements reported by dynamic slicing for $N \in \{5, 10, 20, 30\}$ (% Errors Localized in Table 3).

Overall, a single faulty statement is ranked within the first quarter of the most suspicious program statements reported by dynamic slicing, on average (*cf. Table 3*). This implies that a developer (using dynamic slicing) inspects only 21% (about 40 LoC) of the executable program statements before locating the fault, on average (i.e., equal to $1 - score$). This performance was independent of the source or type of the errors (i.e., real or seeded errors). Dynamic slicing was particularly highly effective in locating faults for errors in CoREBench and errors in IntroClass, where it ranks the faulty statements within the top 15% (81 LoC) and 17% (3 LoC) of the program statements, respectively (*cf. Table 3*).

Table 4: Statistical Debugging Effectiveness on Single Faults. Best scores for each (sub)category are in **bold**; higher scores are better. For instance, Kulczynski2 is the best performing (single bug optimal) formula for *all programs* (0.737), on average.

SBFL Family	Formula	SIR	Intro Class	Code flaws	Core bench	Average	
						(Bugs)	(Prog.)
Popular	Tarantula	0.78	0.76	0.70	0.79	0.709	0.732
	Ochiai	0.83	0.76	0.69	0.79	0.709	0.735
	Jaccard	0.80	0.76	0.69	0.79	0.702	0.728
Human Generated	Naish_1	0.83	0.74	0.69	0.79	0.710	0.733
	Naish_2	0.81	0.74	0.69	0.79	0.709	0.731
	Russel_Rao	0.67	0.59	0.57	0.77	0.602	0.611
	Binary	0.69	0.59	0.57	0.77	0.603	0.614
	Wong_1	0.67	0.59	0.57	0.77	0.602	0.611
	D^2	0.73	0.62	0.56	0.80	0.598	0.618
	D^3	0.75	0.62	0.56	0.80	0.601	0.622
GP Evolved	GP_02	0.75	0.72	0.66	0.69	0.668	0.688
	GP_03	0.77	0.68	0.63	0.63	0.643	0.663
	GP_13	0.81	0.74	0.69	0.79	0.709	0.731
	GP_19	0.56	0.69	0.65	0.75	0.631	0.649
Single Bug Optimal	PattSim_2	0.85	0.68	0.69	0.76	0.705	0.721
	lex_Ochiai	0.83	0.74	0.69	0.79	0.710	0.733
	m9185	0.83	0.74	0.70	0.79	0.715	0.735
	Kulczynski2	0.83	0.76	0.70	0.79	0.713	0.737

For all programs, dynamic slicing reports the faulty statement within the top 21% (40 LoC) of the most suspicious statements, on average.

A developer or tool using dynamic slicing will locate the faulty statement after inspecting only a handful of suspicious statements. In our evaluation, for most errors, the faulty statement can be identified after inspecting only five to ten most suspicious statements reported by dynamic slicing. Specifically, the faulty statement is ranked within the top five to ten most suspicious statements for 62% to 78% of all errors, respectively (*cf. Table 3*). Notably, a developer will locate the faulty statement for 55% of artificial errors and 92% of real errors if she inspects only the top 20 most suspicious statements. Overall, most programs (85%) can be debugged by inspecting the top 30% (58 LoC, on average) of the statements reported by dynamic slicing. These results demonstrate the high effectiveness of dynamic slicing in fault localization.

Dynamic slicing reports a single faulty statement within the top 5–10 most suspicious statements for most errors (62% to 78%, respectively).

RQ2 Effectiveness of Statistical Debugging

Which statistical formula is the most effective at fault localization? First, we investigate the effectiveness of 18 statistical formulas using four benchmarks containing 369 errors (*cf. Table 1*). To determine the most effective statistical formula, for each formula, we examined the proportion of statements a developer would *not need* to inspect after locating a single faulty statement

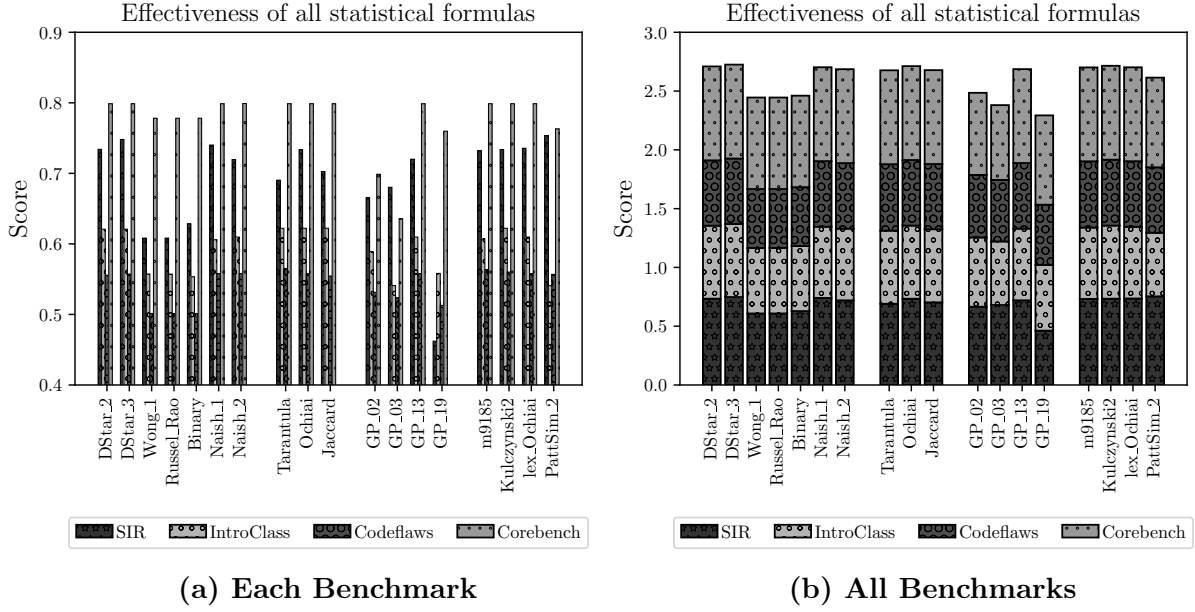


Figure 33: Effectiveness of each statistical debugging formula: (a) results are grouped into bars for each family, and (b) cumulative results for all benchmarks (stacked for all families)

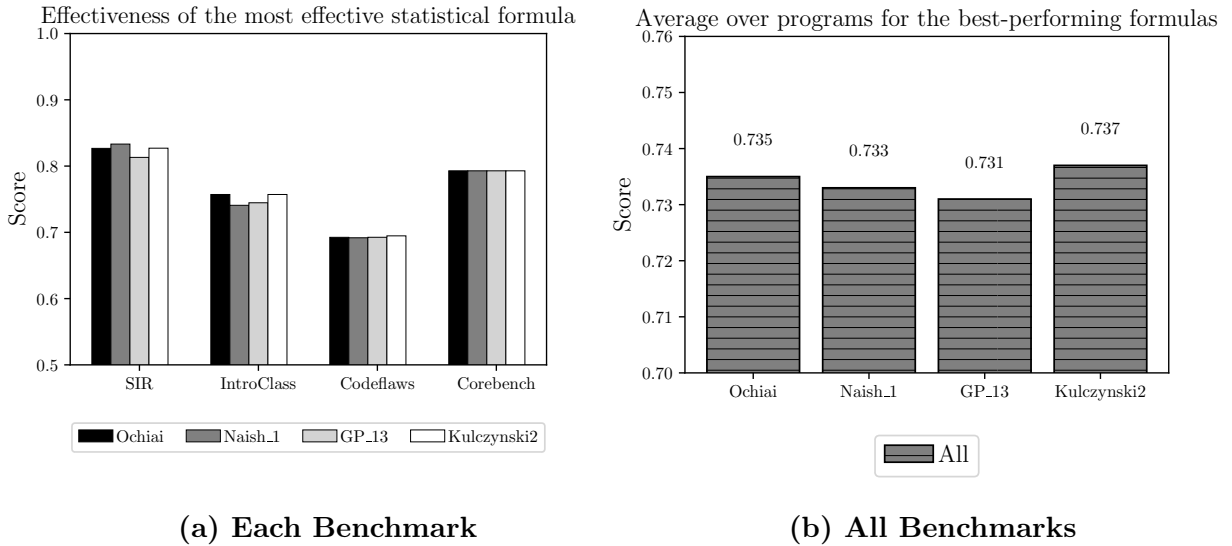


Figure 34: Effectiveness of the most effective statistical formula in (a) each family and (b) the overall average

Table 5: Effectiveness of Kulczynski2 on single faults, i.e. the most effective statistical formula

Benchmark	Score	% Errors Localized if developer inspects N most suspicious LoC			
		5	10	20	30
IntroClass	0.76	80.00	85.00	100	100
Codeflaws	0.69	64.37	86.23	97.57	99.19
CoREBench	0.79	22.22	25.93	37.04	48.15
Real	0.72	61.56	80.61	92.18	94.56
Artificial (SIR)	0.83	35.14	41.89	68.92	71.62
Avg. (Bugs)	0.713	56.25	72.83	87.50	89.95

(*score* in Table 4). Figure 33 (a) and (b) further illustrate the effectiveness of the SBFL formulas. Then, for the best performing statistical formula, we inspected the percentage of errors for which a developer can effectively locate the faulty statement, if she inspects only N most suspicious statements for $N \in \{5, 10, 20, 30\}$ (*% Errors Localized* in Table 5).

Overall, the *single bug optimal* formulas are *the most effective family of statistical formulas*, they are the best performing formulas across all errors and programs. In particular, on average, **PattSim2** performed best for injected errors (i.e. **SIR**), while **Kulczynski2** outperformed all other formulas for real errors, especially for **IntroClass** (*cf. Table 4*). **Bold** values in Table 4 indicate the best performing formula for each family and (sub)category. For instance, **Kulczynski2** is the best performing (single bug optimal) formula for *all programs* (0.737). The performance of single-bug optimal formulas supports the results obtained in previous works [195]. This family of statistical formulas are particularly effective because they are optimized for programs containing a single bug; based on the observation that if a program contains only a single bug, then all failing traces cover that bug [196].

The single bug optimal statistical formulas outperformed all other SBFL formulas, for both injected and real errors, on average.

The *most effective statistical formula* is **Kulczynski2**, it outperformed all other formulas in our evaluation (*see Table 4 and Figure 33 (a) and (b)*). The most effective statistical formula for each family are **Ochiai**, **Naish_1**, **GP_13** and **Kulczynski2** for the *popular*, *human-generated*, *genetically-evolved* and *single bug optimal* families, respectively. Figure 34 (a) and (b) compares the performance of the most effective formula in each family. For instance, in the *popular* statistical family, **Ochiai** is the best performing formula, both for *all errors* (0.709) and *all programs* (0.735) (*cf. Table 4*). Meanwhile, in the single bug optimal family, **Kulczynski2** is the best performing formula for *all programs* (0.737) (*cf. Table 4*).

Indeed, a developer using **Kulczynski2** will inspect the least number of suspicious program statements before finding the faulty statement. On average, **Kulczynski2** required a developer to inspect about 26% (51 LoC) of the program code before finding the faulty statement. Among all statistical formulas, it has the highest suspiciousness rank for 40% (14 out of 35) of the programs and 72% (265 out of 369) of all errors. It is also the most effective statistical formula for localizing real errors.⁴⁰

⁴⁰Further evaluations in this chapter use **Kulczynski2** as the default “*statistical debugging*” formula.

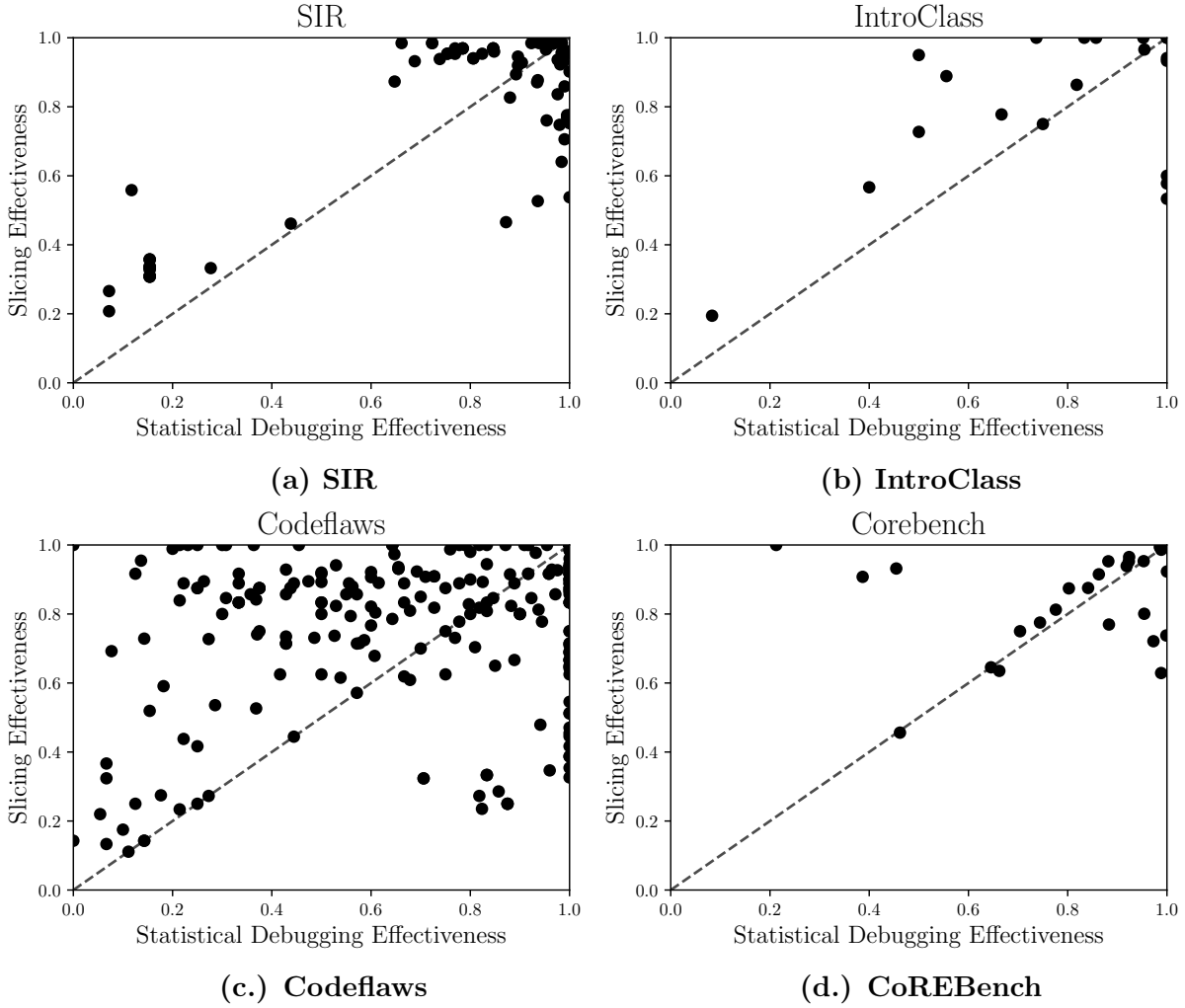


Figure 35: Direct comparison of fault localization effectiveness between statistical debugging (Kulczynski2) and dynamic slicing (on single faults) in each benchmark

Kulczynski2 is the most effective statistical formula, on average, requiring a developer to inspect only 26% (51 LoC) before finding the fault.

A tool or developer using *Kulczynski2* will locate the faulty statement after inspecting five to ten most suspicious statements. The faulty statement is ranked within the top five to ten most suspicious statements for most errors, i.e. 56% to 73% of all errors, respectively (*cf. Table 5*). Overall, most programs (60%) can be debugged by inspecting the top 30% (58 LoC) of the suspicious statements reported by *Kulczynski2*.

Kulczynski2 reports the faulty statements within the top 5–10 most suspicious statements for 56% to 72% of all errors, respectively.

Is the difference in the performance of Kulczynski2 statistically significant, in comparison to the best performing formula for each statistical debugging family? In our evaluation, the difference in the performance of *Kulczynski2* (i.e. the best performing formula) is not statistically significant. Table 6 highlights the statistical tests comparing *Kulczynski2* to the best performing statistical formula in each family, i.e. *Kulczynski2* vs. {*Ochiai*, *Naish1*, *GP13*}. Notably, the performance

Table 6: Statistical Tests for the most effective Statistical Debugging Formulas; *Odds ratio ψ* (all ratios are statistically significant *Mann-Whitney U-test* < 0.05 for all tests)

Benchmark	Odds Ratio ψ (Mann Whitney test score U)		
	Kulczynski2 vs. Ochiai	Kulczynski2 vs. Naish1	Kulczynski2 vs. GP_13
SIR	0.2985 (0.0002)	0.0004 (0)	0.0004 (0)
IntroClass	0.0006 (0)	0.0183 (0)	0.0059 (0)
Codeflaws	0.0013 (0)	0.0005 (0)	0.0002 (0)
CoREBench	0.0003 (0)	0.0003 (0)	0.0003 (0)
All Bugs	0.0106 (0)	0.0006 (0)	0.0002 (0)

of Kulczynski2 is not statistically significant, in comparison to the the best statistical formula for each family. This is evident from the fact that the *odds ratio is less than one* ($\psi < 1$) for all test comparisons (see Table 6). This suggests that Kulczynski2 has no statistically significant advantage over the best performing statistical formulas in each family; despite the fact that, in absolute terms, Kulczynski2 outperforms the best formula in each family.

Kulczynski2 has no statistically significant advantage over the best formula in other SBFL families (i.e., Ochiai, Naish1 and GP13).

RQ3 Comparing Statistical Debugging and Dynamic Slicing

How effective is the most effective statistical formula in comparison to dynamic slicing? We compare the performance of the most effective statistical formula (Kulczynski2) to that of dynamic slicing (cf. Figure 35 and Figure 36).

We find that on average, *dynamic slicing is more effective than statistical debugging at fault localization. Slicing is about eight percentage points more effective than the best performing statistical formula* for all programs in our evaluation (cf. Figure 36, Table 3 and Table 5). For all errors in our study, a programmer using dynamic slicing needs to examine about three-quarters (78%) of those statements that she would need to examine if she used statistical debugging.⁴¹ This result is independent of the type of errors or program. Figure 36 shows that dynamic slicing consistently outperforms statistical debugging for each benchmark, with slicing consistently localizing all faults ahead of statistical debugging.

Overall, dynamic slicing was eight percentage points more effective than the best performing statistical debugging formula, i.e. Kulczynski2.

For two-third of bugs (66%, 243 out of 369 errors), dynamic slicing will find the fault earlier than the best performing statistical debugging formula. Figure 35 shows a direct comparison of the scores computed for slicing and statistical debugging. Each scatter plot shows for each error the effectiveness score of statistical debugging on the x-axis and the effectiveness score of slicing on the y-axis. Errors plotted above the diagonal line are better localized using dynamic slicing. For all benchmarks, the majority of the points are above the diagonal line which indicates

⁴¹Percentage improvement is measured as $\frac{1-0.794}{1-0.737}$. Note that *score* by itself gives the number of statements that need *not* be examined.

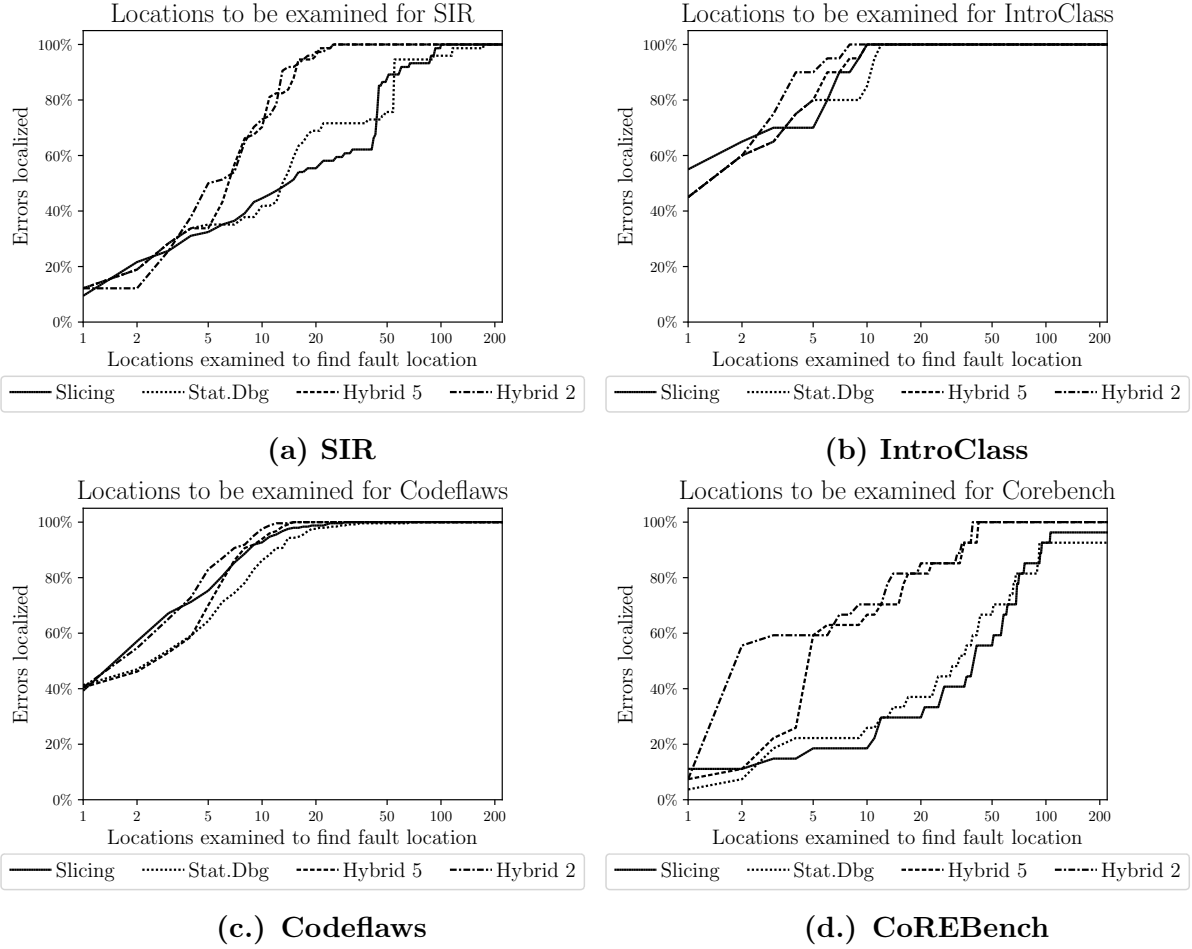


Figure 36: Cumulative frequency of the locations to be examined, for dynamic slicing vs. statistical debugging vs. the hybrid approach (on single faults) in each benchmark

that slicing outperforms statistical debugging in most cases. We can see that dynamic slicing consistently outperforms statistical debugging across all benchmarks.

For two-third (66%) of bugs, dynamic slicing locates the fault earlier than the best performing statistical debugging formula, i.e. Kulczynski2.

To compare the significance of dynamic slicing and statistical debugging, we compute the odds ratio and conduct a Mann-Whitney U -test (cf. *Slicing vs. Kulczynski2* in Table 7). The odds ratio is in favor of dynamic slicing ($\psi > 1$) for all projects. In particular, slicing is 62% more likely to find a faulty statement earlier than statistical debugging, this likelihood is also statistically significant according to the Mann-Whitney test. The *statistically significant odds ratio* is explained by slicing being more effective than statistical debugging in most cases. For instance, slicing is more effective than statistical debugging for 50 out of 74 bugs in the SIR benchmark and for 18 out of 27 bugs in CoREBench.

Dynamic slicing is significantly more likely to find a faulty statement earlier than statistical debugging.

Table 7: Statistical Tests for all three Fault Localization Techniques: Odds ratio ψ (Mann-Whitney U -test p-values (U) are in brackets), odds ratio with statistically significant p-values determined by Mann-Whitney (U -test) are in **bold**

Benchmark	Odds Ratio ψ (Mann whitney test score)		
	Slicing vs. Kulczynski2	Slicing vs. Hybrid-2	Hybrid-2 vs. Kulczynski2
SIR	4.25 (0.0000)	0.81 (0.2568)	5.44 (0.0000)
IntroClass	2.16 (0.1087)	0.68 (0.2713)	0.68 (0.2713)
Codeflaws	1.16 (0.2094)	0.41 (0.0000)	1.05 (0.3938)
CoREBench	2.06 (0.0904)	0.06 (0.0000)	16.74 (0.0000)
All Bugs	1.62 (0.0006)	0.42 (0.0000)	1.69 (0.0002)

RQ4 Sensitiveness of the Hybrid Approach

How many suspicious statements (reported by statistical debugging, i.e. Kulczynski2) should a tool or developer inspect before switching to slicing? We examine the sensitiveness of the hybrid approach to varying absolute values of N . We evaluate how the number of suspicious statements inspected before switching to slicing influences the effectiveness of the hybrid approach. In particular, we investigated the effect of N values (2, 5, 10, 15, 20) on the performance of the hybrid approach, in order to determine the optimal number of suspicious statements to inspect before switching to slicing.

A programmer that switches to slicing after investigating the top five most suspicious statements can localize more errors than if switching after investigating more suspicious statements. Figure 37 shows the impact of other values of N on the effectiveness of the hybrid approach. Note that the hybrid approach degenerates to dynamic slicing when $N = 0$ and to statistical debugging when N is large (e.g., program size). We see that Kochhar’s suggestion of $N = 5$ is a good value for our subjects, in particular inspecting at most five statements before switching to slicing outperforms both slicing and statistical debugging. As we see in Figure 38, the hybrid approach with $N = 2$ and $N = 5$ outperforms both slicing and statistical debugging (Kulczynski2). Hence, *a developer is most effective if she inspects at most five most suspicious statements reported by statistical debugging before switching to slicing.*

A tool using our hybrid approach is most effective when inspecting only the top two most suspicious statements ($N = 2$) reported by statistical debugging, before switching to slicing. Hence, we recommend the use of the hybrid approach (with $N = 2$) for fault localization, and at most five suspicious statements should be inspected before switching to slicing.⁴²

The hybrid approach is most effective when a programmer inspects at most two statements ($N = 2$) before switching to slicing.

RQ5 Effectiveness of the Hybrid Approach

Which technique is the most effective in fault localization? Which technique is more likely to find fault locations earlier? We now investigate the effectiveness of the hybrid approach, in comparison to slicing and statistical debugging. First, we examine the number of program statements that

⁴²We use the best values of N (i.e. $N = 2$ and $N = 5$) to for the rest of our evaluation.

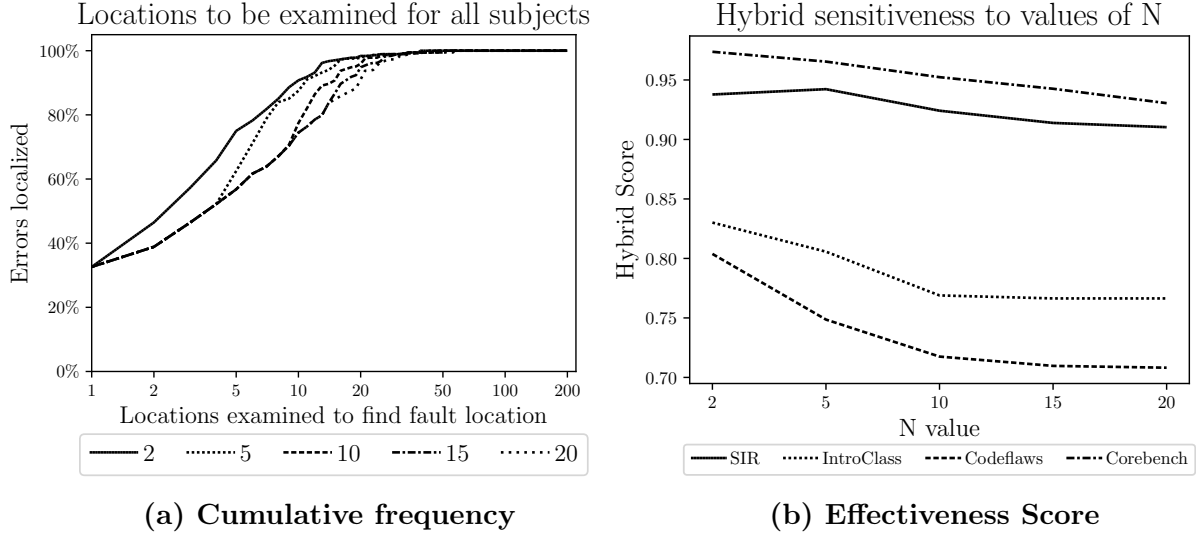


Figure 37: Hybrid sensitiveness to different values of $N \in \{2, 5, 10, 15, 20\}$ showing (a) the cumulative frequency of locations to be examined for all errors (left), and (b) the effectiveness score for each benchmark using the hybrid approach (right).

Table 8: Effectiveness of the Hybrid approach with $N = 2$

Benchmark	Score	% Errors Localized if developer inspects N most suspicious LoC			
		5	10	20	30
IntroClass	0.83	90.00	100	100	100
Codeflaws	0.80	83.00	97.57	100	100
CoREBench	0.97	59.26	70.37	85.19	85.19
Real	0.83	81.29	95.24	98.64	98.64
Artificial (SIR)	0.94	50.00	72.97	97.30	100
Avg. (Bugs)	0.844	75.00	90.76	98.37	98.91

need to be inspected to localize all faults for each technique (Figure 38), as well as the absolute effectiveness score of each technique (Table 3, Table 5 and Table 8). Then, we evaluate the likelihood of each technique to find the fault locations earlier than the other two techniques (Table 7).

Notably, if the programmer is willing to inspect no more than 20 statements, the hybrid approach will localize the fault location for almost all (98%) of the bugs (*cf. Table 8 and Figure 38*). In contrast, both statistical debugging and slicing can only localize almost all (98%) faults after inspecting about five times as many statements, i.e. 100 LoC. In fact, if the programmer inspects only 20 LoC, slicing and statistical debugging would only find the fault location for about 85% and 88% of the bugs, respectively.

The hybrid approach localizes the fault location for almost all (98%) of the bugs after inspecting no more than 20 LoC.

In absolute numbers, the hybrid approach is the most effective fault localization technique, followed by slicing, which is more effective than statistical debugging (see Table 8, Figure 36 and Figure 38). The hybrid approach ($N = 2$) is about seven percentage points more effective than slicing, and about fifteen percentage points more effective than statistical debugging (*cf. Table 3, Table 5 and Table 8*). Overall, it improves the performance of both slicing and statistical

debugging. For instance, a programmer using the hybrid approach needs to examine only about half (58%) and three-quarter (75%) of those statements that she would need to examine if she used slicing and statistical debugging, respectively.

The hybrid approach is significantly more effective than slicing and statistical debugging, respectively.

We compute the odds ratio and conduct a Mann-Whitney U -test, in order to determine the significance of the hybrid approach. The odds ratio for all projects is strictly in favor of the hybrid approach ($\psi > 1$ in Table 7). Specifically, the hybrid approach is (69%) more likely to find a faulty statement earlier than statistical debugging (cf. “Hybrid-2 vs. Kulczynski2” in Table 7). Moreover, a programmer is (42%) less likely to find the fault location early if she localizes with dynamic slicing instead of the hybrid approach (cf. “Slicing vs. Hybrid-2” in Table 7).

The *statistically significant odds ratio* is explained by the hybrid approach being more effective than slicing and statistical debugging in most cases. *The majority of bugs is best localized by the hybrid approach.* For more than half of the bugs (56%, 208 out of 369 errors), the hybrid approach will find the fault earlier than both slicing and statistical debugging. In particular, for CoREBench, the hybrid approach is more effective than both techniques for 19 out of 27 bugs, as well as for 33 out of 74 bugs in SIR.

The hybrid approach is significantly more likely to find a faulty statement earlier than dynamic slicing and statistical debugging.

RQ6 Real Errors vs. Artificial Errors

In this section, we evaluate the effect of error type on the effectiveness of an AFL technique, in particular, the difference between evaluating AFL techniques on artificial errors (i.e., SIR⁴³) versus real errors (i.e., IntroClass, Codeflaws and CoREBench). We examine the performance of each technique on each error type and portray the bias and differences in such evaluations. Figure 35 and Figure 36 highlight the difference between evaluating a fault localization technique on real or artificial errors. Table 9 summarizes the difference in the effectiveness of each AFL technique when using real or artificial faults. Table 3, Table 5 and Table 8 also quantify the difference in the effectiveness of all three AFL techniques (i.e., dynamic slicing, statistical debugging and the hybrid approach, respectively) on real and artificial faults.

What is the most effective statistical debugging formula for artificial or real faults? In our evaluation, the error type influences the effectiveness of a statistical debugging formula. PattSim_2 is the most effective statistical formula for artificial faults ($score=0.85$), this is closely followed by Ochiai and Naish_1 with $score = 0.83$ (see Table 4). Meanwhile, for real faults, the most effective statistical formulas are Kulczynski2 and Tarantula with scores 0.76, 0.70 and 0.79 for IntroClass, Codeflaws and CoREBench, respectively (see Table 4).⁴⁴Notably, the most effective

⁴³The SIR benchmark is the *most used subject for the evaluation of AFL techniques*, especially statistical fault localization [172].

⁴⁴Even among real faults, the most effective formula depends on the benchmark. For instance the DStar algorithm slightly outperforms Kulczynski2 and Tarantula on CoREBench, despite performing worse on IntroClass and Codeflaws.

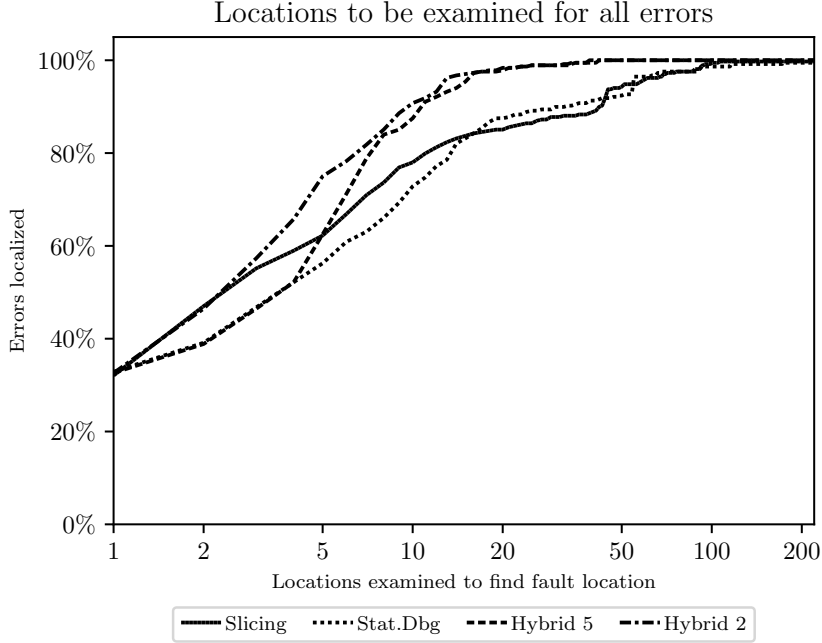


Figure 38: Cumulative frequency of the locations to be examined for the hybrid approach, in comparison to statistical debugging (Kulczynski2) and dynamic slicing, for all (Single) Faults. Inspecting only the top two suspicious code locations, Hybrid-2 and dynamic slicing perform similarly (localizing about 47% of errors each); they outperform statistical debugging (39% of errors localized). However, inspecting only the top five locations, Hybrid-2 clearly outperforms slicing and dynamic slicing by localizing 75% of errors, while slicing performs better than statistical debugging (62% vs. 56% of errors).

Table 9: Real vs. Artificial faults

Benchmark	Percentage (# LoC)		
	Statements Inspected before locating fault		
	Slicing	Kulczynski2	Hybrid
IntroClass	17% (3)	24% (4)	17% (3)
Codeflaws	23% (4)	31% (5)	20% (3)
CoREBench	15% (80)	21% (112)	3% (14)
Real	18% (29)	25% (40)	13% (7)
Artificial (SIR)	21% (31)	18% (26)	6% (9)
All Bugs	21% (40)	26% (51)	15% (30)

formula for artificial faults is different from the most effective formula for real faults. This implies that the error type can influence the performance of an AFL technique. Thus, we recommended to always evaluate debugging aids using real faults.

The effectiveness of a statistical debugging formula depends on the error type: the most effective formula differs for artificial (PattSim_2) and real faults (Kulczynski2 and Tarantula).

How does the effectiveness of statistical debugging compare to that of dynamic slicing, for artificial and real faults? On one hand, dynamic slicing performs worse than statistical debugging on artificial faults (SIR): A developer (or tool) has to inspect 21% of the program to find the fault, in contrast to 18% for Kulczynski2, on average (see Table 9). On the other hand, dynamic

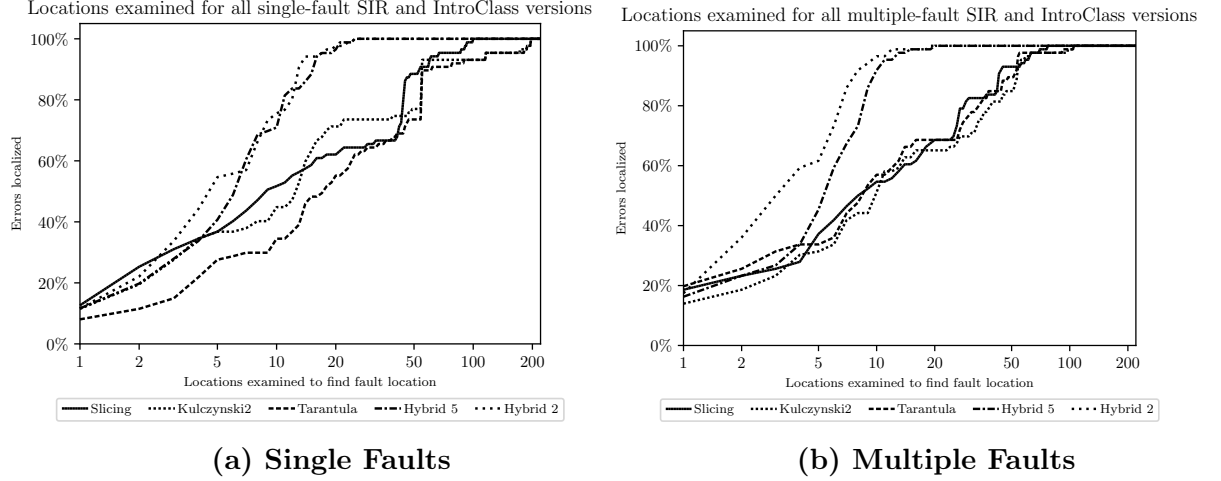


Figure 39: Cumulative frequency of the locations to be examined for (a) single-fault and (b) multiple-fault versions of the **SIR** and **IntroClass**, using the hybrid approach, statistical debugging (Kulczynski2 and Tarantula) and dynamic slicing

slicing performs better than statistical debugging on real faults (i.e., **IntroClass**, **Codeflaws** and **CoREBench**). For real errors, a developer has to inspect (7%) less statements when using dynamic slicing (18%) compared to slicing (25%). Again, this shows that the error type has a significant influence on the effectiveness of an AFL technique.

Statistical debugging performs better on artificial faults, while dynamic slicing performs better on real faults.

What is the most effective AFL approach on artificial and real faults? The hybrid approach is the most effective AFL approach, outperforming both dynamic slicing and statistical debugging (see Table 9). In particular, depending on the error type, a developer or tool using the hybrid approach inspects only one-third to less than three-quarter (0.3 to 0.7) of the statements inspected when using dynamic slicing or statistical debugging. This shows that the effectiveness of the hybrid approach is independent of error type.

The hybrid approach is the most effective approach, regardless of error type, i.e. artificial or real faults.

We observed that fault localization effectiveness on artificial errors does not predict results on real faults. In our evaluation, the performance of dynamic slicing and statistical debugging are different depending on the error type. For instance, Table 9 clearly shows that dynamic slicing performs better on real faults, while statistical debugging performs better on artificial faults. This result illustrates that the performance of an AFL technique on artificial faults is not predictive of its performance in practice. Hence, it is pertinent to evaluate AFL techniques on real faults rather than artificial faults, this is in line with the findings of previous studies [194].

The effectiveness of an AFL technique on artificial faults does not predict its effectiveness on real faults.

Table 10: Effectiveness of all AFL techniques on Single and Multiple Faults for **SIR** and **IntroClass** benchmarks. Single Fault Scores are in italics and bracketed, i.e. (*Single*), while Multiple Fault Scores are in normal text. For multiple faults, the best scores for each (sub)category are in **bold**; higher scores are better. For instance, **Tarantula** is the best performing (popular) statistical debugging formula for *all programs with multiple faults with score 0.8269*, on average.

AFL Technique	Formula/ Approach	SIR MULT (<i>Single</i>)	IntroClass MULT (<i>Single</i>)	Average (Prog.) (Vers.)	
Popular SBFL	Tarantula	0.8214 (<i>0.6907</i>)	0.8324 (<i>0.7464</i>)	0.8269 (<i>0.7186</i>)	0.7878 (<i>0.6266</i>)
	Ochiai	0.7796 (<i>0.7337</i>)	0.7747 (<i>0.7464</i>)	0.7772 (<i>0.7401</i>)	0.7560 (<i>0.6514</i>)
	Jaccard	0.7720 (<i>0.7029</i>)	0.7747 (<i>0.7464</i>)	0.7734 (<i>0.7247</i>)	0.7532 (<i>0.6342</i>)
Human Generated SBFL	Naish_1	0.7215 (<i>0.7399</i>)	0.7638 (<i>0.7448</i>)	0.7426 (<i>0.7423</i>)	0.7136 (<i>0.6682</i>)
	Naish_2	0.7484 (<i>0.7207</i>)	0.7747 (<i>0.7464</i>)	0.7615 (<i>0.7336</i>)	0.7404 (<i>0.6596</i>)
	Russel_Rao	0.7350 (<i>0.6088</i>)	0.7586 (<i>0.6394</i>)	0.7468 (<i>0.6241</i>)	0.7185 (<i>0.6051</i>)
	Binary	0.7125 (<i>0.6283</i>)	0.7553 (<i>0.6378</i>)	0.7339 (<i>0.633</i>)	0.6975 (<i>0.6129</i>)
	Wong_1	0.7350 (<i>0.6088</i>)	0.7586 (<i>0.6394</i>)	0.7468 (<i>0.6241</i>)	0.7185 (<i>0.6051</i>)
	D^2	0.7718 (<i>0.7345</i>)	0.7747 (<i>0.7464</i>)	0.7732 (<i>0.7404</i>)	0.7553 (<i>0.6523</i>)
	D^3	0.7680 (<i>0.7484</i>)	0.7747 (<i>0.7464</i>)	0.7714 (<i>0.7474</i>)	0.7553 (<i>0.6611</i>)
GP Evolved SBFL	GP_02	0.7633 (<i>0.6725</i>)	0.7747 (<i>0.7157</i>)	0.7690 (<i>0.6941</i>)	0.7498 (<i>0.6133</i>)
	GP_03	0.7473 (<i>0.6813</i>)	0.7747 (<i>0.6169</i>)	0.7610 (<i>0.6491</i>)	0.7402 (<i>0.6285</i>)
	GP_13	0.7456 (<i>0.7211</i>)	0.7747 (<i>0.7464</i>)	0.7602 (<i>0.7338</i>)	0.7398 (<i>0.6606</i>)
	GP_19	0.7629 (<i>0.4673</i>)	0.7558 (<i>0.6237</i>)	0.7593 (<i>0.5455</i>)	0.7279 (<i>0.4876</i>)
Single Bug Optimal SBFL	PattSim_2	0.7608 (<i>0.7537</i>)	0.7747 (<i>0.6544</i>)	0.7677 (<i>0.704</i>)	0.7301 (<i>0.6506</i>)
	lex_Ochiai	0.7532 (<i>0.7356</i>)	0.7747 (<i>0.7464</i>)	0.7640 (<i>0.741</i>)	0.7396 (<i>0.6646</i>)
	m9185	0.8183 (<i>0.7570</i>)	0.8315 (<i>0.7225</i>)	0.8249 (<i>0.7397</i>)	0.7664 (<i>0.6646</i>)
	Kulczynski2	0.7885 (<i>0.7572</i>)	0.7747 (<i>0.7464</i>)	0.7816 (<i>0.7518</i>)	0.7588 (<i>0.6689</i>)
Program Slicing	Dynamic Slicing	0.8357 (<i>0.7935</i>)	0.5487 (<i>0.8602</i>)	0.6922 (<i>0.8269</i>)	0.7840 (<i>0.7535</i>)
Hybrid Approach	Hybrid-2	0.9627 (<i>0.9358</i>)	0.9057 (<i>0.888</i>)	0.9342 (<i>0.9119</i>)	0.9457 (<i>0.9237</i>)
	Hybrid-5	0.9505 (<i>0.9397</i>)	0.8406 (<i>0.814</i>)	0.8955 (<i>0.8768</i>)	0.9206 (<i>0.8974</i>)

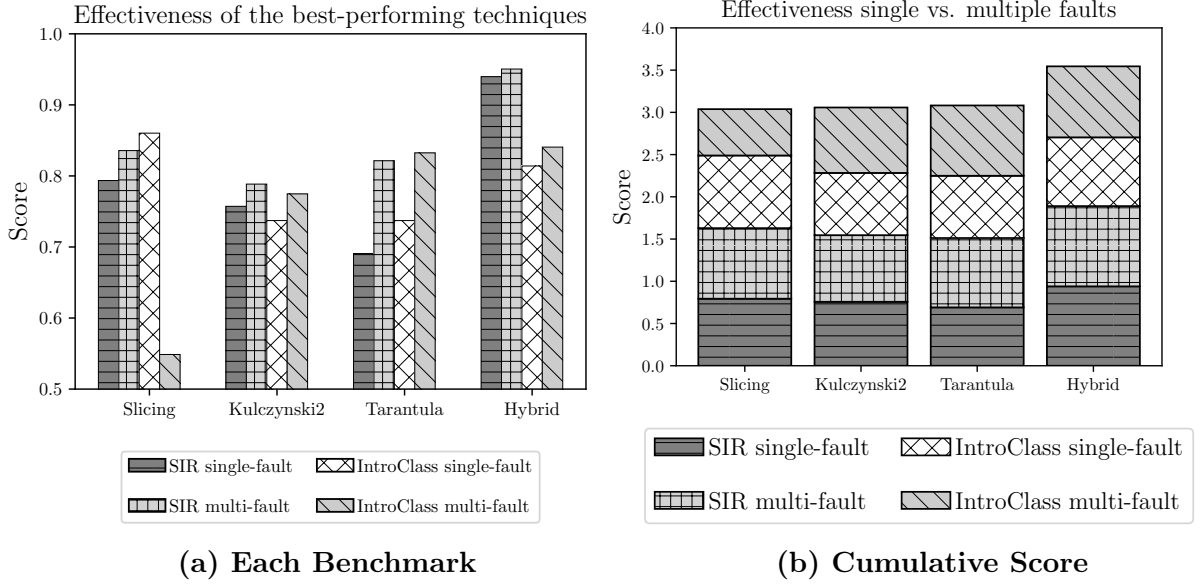


Figure 40: Effectiveness of each technique for Single and Multiple Fault(s) in SIR and IntroClass: (a) Scores for each benchmark and (b) Scores for both benchmarks

RQ7 Single Fault vs. Multiple Faults

In this section, we compare the effectiveness of all three AFL techniques on programs with *multiple faults*. Then, we examine the effect of multiple faults on the performance of each technique and the difference between evaluating an AFL technique on single or multiple fault(s). In this experiment, we employ the original single-fault versions of the **SIR** and **IntroClass** benchmarks, as well as the multiple-fault versions of the same benchmarks, called **SIR-MULT** and **IntroClass-MULT**, respectively. Table 10 highlights the results for single and multiple fault(s) for all AFL techniques, including statistical debugging, hybrid and dynamic slicing. Figure 39, Figure 40 and Figure 41 illustrate the difference in the performance of each technique when given programs with a single fault or multiple faults.

What is the most effective statistical debugging formula for multiple faults? In our evaluation, the most effective statistical debugging formula for multiple faults is **Tarantula** (0.8269), from the *popular* statistical debugging family. It outperforms the other statistical debugging formulas (*cf. Table 10 and Figure 40*). For the other statistical debugging families, the most effective formula for multiple faults are **DStar** (D^2 and D^3), **GP02** and **m9185** for the *popular*, *human-generated* and *genetically evolved* families, respectively (*cf. Table 10*). The performance of **Tarantula** is closely followed by that of the single-bug optimal formulas **m9185** (0.8249). However, the difference in the performance of **m9185** and **Tarantula** is not statistically significant, i.e. $\psi < 1$ (*odds ratio* $\psi = 0.14$, *Mann-Whitney U-test p-value* $U = 0$). Notably, the most effective single-bug optimal formulas (i.e. **m9185**) outperformed the human-generated and genetically evolved formulas (*cf. Table 10*). This illustrates that *single bug optimal* formulas are also effective for multiple faults, despite being specialized for single faults.

***Tarantula** is the most effective statistical debugging formula for multiple faults;
it outperforms all other statistical debugging formulas.*

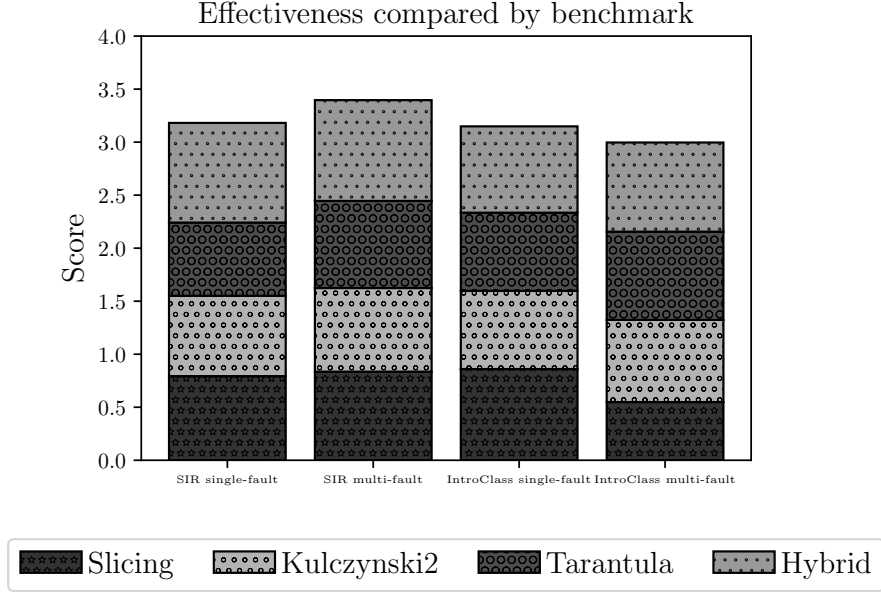


Figure 41: Effectiveness of each technique on Single and Multiple Faults compared by benchmark SIR and IntroClass

For multiple faults, how does the effectiveness of statistical debugging (*Tarantula*) compare to that of dynamic slicing and hybrid? *Tarantula* performs better than dynamic slicing (0.8269 vs. 0.6922) for multiple faults, our results show that the effectiveness of slicing is 16% worse than that of *Tarantula* on multiple faults in the SIR-MULT and IntroClass-MULT programs (see Table 10). This is despite the fact that dynamic slicing (0.8269) outperforms *Tarantula* (0.7186) by 15% on single fault programs (i.e., in SIR and IntroClass benchmarks). Indeed, there is a 13% decrease in the performance of dynamic slicing on multiple faults. This is evident in Figure 40 (a) where the performance of dynamic slicing drops for multiple faults for IntroClass-MULT. This shows that it is beneficial for an AFL technique to employ coverage data from (numerous) failing test cases when diagnosing programs with multiple faults. As expected, it is more difficult for dynamic slicing to diagnose multiple faults: Since a dynamic slice is constructed for only a single failing test case, it is difficult to account for the effect of multiple faults. Overall, the performance of the hybrid approach remains superior to that of dynamic slicing and statistical debugging, regardless of the number of faults present in the program (cf. Table 10, Figure 39 and Figure 40).

*Statistical debugging performs better on multiple faults:
Tarantula is 19% more effective than dynamic slicing on multiple faults.*

Given single or multiple faults, does the effectiveness of an AFL technique improve or worsen? Figure 40 (a) illustrates the difference in the performance of all techniques for single and multiple faults. Results show that all techniques (except dynamic slicing) perform better on multiple faults in comparison to single faults, improvements range from two to 11 percentage points. Notably, *Tarantula*'s performance improved by 11% on multiple faults. Meanwhile, other approaches improved by two to three percentage points, in particular, the hybrid approach, Kulczynski2 and DStar (D^2 and D^3). This illustrates that most AFL approaches (, especially SBFL) perform better on multiple faults than single faults (see Figure 40 (a) and Figure 41).

*Statistical debugging is better suited for diagnosing multiple faults,
while dynamic slicing is more effective at localizing single faults.*

Generally, we found that the performance of a technique on programs containing single faults does not predict its performance on multiple faults. For instance, although `Kulczynski2` outperformed the other statistical formulas for single faults (0.7518), it is outperformed by `m9185` for multiple faults (0.7816 vs. 0.8249) (*cf. Table 10*). This result is also evident from Figure 40 (a) and Table 10, where dynamic slicing outperforms statistical debugging (`Kulczynski2`) for `SIR` single faults (0.8269 vs. 0.7518), but statistical debugging (`m9185` and `Kulczynski2`) clearly outperform dynamic slicing for multiple faults. These results suggest that the number of faults in the program influences the effectiveness of an AFL technique.

*The effectiveness of an AFL technique on single faults does not
predict its effectiveness on multiple faults.*

4.5 Threats to Validity

We discuss the threats to validity for this fault localization study within the framework of [180].

External Validity: refers to the extent to which the reported results can be generalized to other objects which are not included in the study. The most immediate threats to external validity are the following:

- *EV.1) Heterogeneity of Proband.* The quality of the test suites provided by the object of analysis may vary greatly which hampers the assessment of accuracy for practical purposes. However, in our study the test suites are well-stocked and maintained. All projects are open source C programs which are subject to common measures of quality control, such as code review and providing a test case with bug fixes and feature additions.
- *EV.2) Faulty Versions and Fault Injection.* For studies involving artificially injected faults, it is important to control the type and number of injected faults. Test cases become subject to accidental fault injection. Some failures may be spurious. However, in our study we also use real errors that were introduced (unintentionally) by real developers. Failing test cases are guaranteed to fail because of the error.
- *EV.3) Language Idiosyncrasies.* Indeed, our objects contain well-maintained open-source C projects with real errors typical for such projects. However, for instance errors in projects written in other languages, like Java, or in commercially developed software may be of different kind and complexity. Hence, we cannot claim generality for all languages and suggest reproducing our experiments for real errors in projects written in other languages as well.
- *EV.4) Test Suites.* The size of a test suite can influence the performance of an AFL technique. Testing strategies that reduce or increase test suite size such as test reduction or test generation methods (i.e. removing tests or generating new tests) have been shown to improve the performance of some AFL techniques [175, 176]. We mitigate the effect of

test suite size by employing projects with varying test suite sizes (ranging from tens to tens of thousands of test cases) as provided by our subject programs. In our evaluation, we do not generate additional tests or remove any tests from the test suite provided by the benchmarks, in order to simulate the typical debugging scenario for the software project.

- *EV.5) Missing Statements in Slices.* Although, there is a risk of discarding faulty statements during program slicing, dynamic slicing rarely miss faulty statements during fault localization [197]. [197] found that dynamic slicing reports the faulty statement in the top-ten most suspicious statement 91% of the time. We further mitigate this risk by first inspecting statements in the dynamic slice before inspecting other executable statements. Thus, dynamic slicing (eventually) finds the faulty statement for all bugs in our evaluation.

Construct Validity: refers to the degree to which a test measures what it claims to be measuring. The most immediate threats to construct validity are the following:

- *CV.1) Measure of Effectiveness.* Conforming to the standard [172], we measure fault localization effectiveness as ranking-based relative wasted effort. The technique that ranks the faulty statement higher is considered more effective. Parnin and Orso find that “programmers will stop inspecting statements, and transition to traditional debugging, if they do not get promising results within the first few statements they inspect” [178]. However, [180] insist that one may question the *usefulness* of fault locators, but measures of ranking-based relative wasted effort are certainly necessary for evaluating their performance, particularly in the absence of the subjective user as the evaluator.
- *CV.2) Implementation Flaws.* Tools that we used for the evaluation process may be inaccurate. Despite all care taken, our implementation of the 18 studied statistical fault localization techniques, or of approximate dynamic slicing, or of the empirical evaluation may be flawed or subject to random factors. However, we make all implementations and experimental results available online for public scrutiny.

4.6 Related Work

Evaluation of Fault Localization Techniques: The effectiveness of various fault localization approaches have been studied by several colleagues, see Wong et al. [172]. Most papers investigated the effects of program, test and bug features on the effectiveness of statistical debugging. Abreu et al. [179] examined the effects of the number of passing and failing test cases on the effectiveness of statistical debugging, they established that the suspiciousness scores stabilize starting from an average six (6) failing and twenty (20) passing test cases. [194] evaluated the difference between evaluating fault localization techniques on real faults versus artificial faults, using two main techniques, namely statistical debugging and mutation-based fault localization. Notably, their evaluation results shows that results on artificial faults do not predict results on real faults for both techniques, and a hybrid technique is significantly better than both techniques. Keller et al. [198] and Heiden et al. [199] evaluated the effectiveness of statistical fault localization on real world large-scale software systems. The authors found that, for realistic large-scale programs, the accuracy of statistical debugging is not suitable for human developers. In fact, the authors

emphasize the obvious need to improve statistical debugging with contextual information such as information from the bug report or from version history of the code lines. In contrast to our work, none of these papers evaluated the fault localization effectiveness of program slicing, nor compare the effectiveness of slicing to that of statistical debugging.

A few approaches have evaluated the effectiveness of dynamic slices in fault localization [173, 174]. In particular, Zhang et al. [174] evaluated the effectiveness of three variants of dynamic slicing algorithms, namely data slicing, full (dynamic) slicing, and relevant slicing. Like our study, the authors found that slicing considerably reduces the number of program statements that need to be examined to locate faulty statements. In contrast to our study, the authors have not empirically compared the performance of slicing to that of statistical debugging.

Improvements of Statistical Fault Localization: Several authors have proposed approaches to improve statistical fault localization (SBFL). Most approaches are focused on reducing the program spectra (i.e. the code coverage information) fed to statistical debugging, sometimes by using delta debugging [200], program slicing [201, 202, 203], test generation [204], test prioritization [205] or machine learning [206, 207]. In particular, some techniques apply program slicing to reduce the program spectra fed to statistical debugging formulas [208, 201, 209, 201, 202, 203]. The popular page rank algorithm has been used to boost statistical debugging effectiveness by estimating the contributions of different tests to re-compute program spectral [205]. Machine learning algorithms (such as learning to rank) also improves the effectiveness of statistical debugging [207, 206]. In addition, BARINEL employs a combination of bayesian reasoning and statistical debugging to improve fault localization effectiveness, especially for programs with multiple faults [187]. BARINEL combines statistical debugging and *model-based diagnosis* (MBD), i.e., logic reasoning over a behavioral model to deduce multiple-fault candidates: The goal is to overcome the high computational complexity of typical MBD. Search-based test generation has also been combined with SBFL, in order to improve the performance of statistical debugging for Simulink models [204]. However, none of these papers localize faults by following control and data dependencies in the program, i.e. they do not directly use program slicing as a fault localization technique.

Zou et al. [206] found that the combination of fault localization techniques improves over individual techniques, the authors recommend that future fault localization techniques should be evaluated in the combined setting. For instance, *slicing hitting set computation* (SHSC) is a combination of model-based debugging and program slicing that has been applied for fault localization [210]. In contrast to our work, SHSC combines slices of faulty variables, which causes undesirable high ranking of statements executed in many test cases [211]. To address this, Hofer and Wotawa also proposed SENDYS – a combination of statistical debugging and SHSC to improve the ranking of faulty statements [211]. The focus of this work is to provide fault locations at a finer granularity than program blocks. In contrast to typical dynamic slicing, SENDYS analyzes the execution information from both passing and failing test cases and uses statistical debugging results as a-priori fault probabilities of single statements in SHSC [211].

4.7 Discussions and Future Work

The empirical results from this chapter shows that *dynamic slicing remains the technique of choice for programmers and debuggers*. Suspicious statements, as produced by statistical debugging, can provide good starting points for an investigation; but beyond the top-ranked statements, following dependencies is much more likely to be effective. We therefore recommend that following dependencies should remain the primary method of fault localization—it is a safe and robust technique that will get programmers towards the goal.

Dynamic slicing as a fault localization technique can be further improved, in order make it more effective and efficient in software practice. Building on the empirical results from our study (in Chapter 4), we consider the following improvements:

Cognitive load. In our investigation, we did not consider or model the *cognitive load* it takes to understand the role of individual statements in context. Since following dependencies in a program is much more likely to stay within same or similar contexts than statistical debugging, where the ranked suspicious lines can be strewn arbitrarily over the code, we would expect dependency-based techniques to take a lead here. The seminal study of Parnin and Orso [178] found that ranked lists of statements are hardly helping human programmers—let us find out which techniques work best for humans.

Alternate search techniques. There are other search strategies along program dependencies (for instance, starting with the input, and progressing forward through a program; starting at some suspicious or recently changed location; or moving along coarse-grained functions first, and fine-grained lines later) that may be even more efficient both in terms of nodes visited as well as from the assumed cognitive load. Again, this calls for more human studies in debugging.

Experimental techniques. For instance, input minimization techniques (such as delta debugging [212]) offer another means to reduce the cognitive load—by systematically narrowing down the conditions under which a failure occurs. The work of Burger and Zeller [213] on minimization of calling sequences with delta debugging showed dramatic improvements over dynamic slicing, reducing “the search space to 13.7% of the dynamic slice or 0.22% of the source code”. In a recent human study, delta debugging “significantly increased programmers efficiency in failure detection, fault localization and fault correction.” [214].

Symbolic techniques. Finally, following dependencies is still one of the simplest methods to exploit program semantics. Applying symbolic execution and constraint solving would narrow down the set of possible faults. Model-based debugging [215] was one of the first to apply this idea in practice; the more recent BUGASSIST work of Jose and Majumdar “quickly and precisely isolates a few lines of code whose change eliminates the error.” [216].

To summarize, the fault localization techniques proposed for future work can profit from wider evaluations and assessments. In addition, they can also be joined and combined similarly to *our hybrid approach* (in Section 4.2). For instance, one could start with suspicious statements as indicated by statistical fault localization, follow dependencies from there, and skip influences

deemed impossible by symbolic analysis. The key challenge of automated fault localization will be to bring the best of the available techniques together in ways that are *applicable* to a wide range of programs and *useful* for real programmers, who must fix their bugs by the end of the day. Finally, we encourage the use of true defects to compare AFL techniques and a willingness to actually compare to the state of the art techniques, as we do in this work. All of our scripts, tools, benchmarks and results are freely available to support scrutiny, evaluation, reproduction and extension:

`https://tinyurl.com/HybridFaultLocalization`

Debugging Failure-Inducing Inputs

This chapter is taken, directly or with minor modifications, from our 2020 ICSE paper *Debugging Inputs* [217]. My contribution in this work is as follows: (I) original idea; (II) partial implementation; (III) evaluation.

“Every failure carries with it the seed of an equal or greater benefit.”

— Napoleon Hill

5.1 Introduction

In the last decade, techniques for automated debugging and repair have seen great interest in research and practice. A recent survey [1] lists more than 100 papers on automatic fault localization and repair. Recently, social networking giant Facebook provided developers with automatically generated repair suggestions for every failure report of its apps [218]. Almost all of these techniques focus on *program code*, attempting to identify possible fault locations in the code and synthesizing fixes for this code. However, when a program fails on some input, it need not be the program code that is at fault. Hardware failures, hardware aging, transmission errors may all cause data to get corrupted. In computer hardware, radiation can impact memory cells, leading to bit flips and again data corruption. And finally, data can be corrupted through software bugs, with the processing software writing out malformed or incomplete data. If data is corrupted, the easiest remedy is to use a backup. But if a backup does not exist (or is too old, or fails to be processed), one may want to recover *as much data as possible* from the existing data—or in other words, *debug the data*.

Some programs come with application-specific means to recover data. Input parsers can recover from syntactical errors by applying sophisticated recovery strategies; in a programming language, this may involve skipping the current statement or function and resuming with the next one [219]. When detecting a corrupted or incomplete file, Microsoft Office programs may attempt to recover from the error, using a number of undisclosed approaches [220]. When a program does *not* implement a good recovery strategy, though, users are left to their own devices, using general-purpose editors to identify file contents and possible corrupted parts.

As listed above, general-purpose automated debugging techniques focus on faults in code and do not provide much help in such situations, as they would regularly identify the input parser

```
{ "item": "Apple", "price": **3.45 }
```

Figure 42: Failing JSON input

```
{
```

Figure 43: Failing input reduced with *ddmin*

```
{ "item": "Apple", "price": 3.45 }
```

Figure 44: Failing input repaired with *ddmax*

and its error-handling code as being associated with the fault. The *delta debugging* (*ddmin*) algorithm [57], however, focuses on identifying error causes in the input; in repeated runs with reduced inputs, it simplifies a failure-inducing input down to a minimum that reproduces the error. Unfortunately, delta debugging is not a good fit: applied to invalid inputs, it produces the smallest subset of the input that also produces an input error—typically a single character. As an example, consider Figure 42, a JSON input with a syntax error; *ddmin* produces the reduced input in Figure 43, consisting of a single `{` character, which also produces a syntax error. This is neither helpful for diagnosis nor a basis for data recovery.

In this chapter, we first investigate the *relevance* of invalid inputs in software practice and collect *empirical evidence on the prevalence and causes of invalid inputs*. Secondly, we introduce a *generic input repair method* that automatically (1) *identifies* which parts of the input data prevent processing, and (2) *recovers* as much of the (valuable) input data as possible. Like *ddmin*, our approach runs the program under test repeatedly with different subsets of the input, assessing whether the subset can be processed or not. Also, it does not need any kind of program analysis and can thus be used in a wide range of settings. Unlike *ddmin*, however, which aims at minimizing the failure-inducing input, our *ddmax* algorithm aims at *maximizing the passing input*. Its result is a subset of the input that (1) can be successfully processed and (2) is *1-maximal*: no further element from the failing input can be added without causing the input to become invalid again.

Applied on our example from Figure 42, *ddmax* produces the “repaired” (passing) input subset in Figure 44, in which the confounding `**` characters (and nothing else) are removed. The difference between the original input (Figure 42) and the repaired input (Figure 44), listed in Figure 45, actually makes a precise diagnosis of the failure cause and can be given to developers for further debugging steps.

Note that while *ddmax* recovers a maximum of *data*, it does not recover a maximum of *information*; in our example, we do not know whether `3.45` actually is the correct price. However, the repaired input can now be read and processed by the program at hand, enabling humans to read and check their document and engage into additional recovery steps.

Although, many applications produce error messages when processing invalid inputs, most error messages are vague. Often, applications simply report that an input is corrupted, without repairing the input or providing the reason for the invalidity. However, *ddmax* identifies the invalid input fragment quickly (for debuggers) while also preserving a maximum of content (for



Figure 45: Difference between failing and repaired input

end users).

The remainder of this chapter makes the following contributions:

An empirical study of invalid inputs in practice. We evaluate the prevalence of invalid input in the wild (Section 5.2). We crawled thousands of input files from *github* and determine the set of valid and invalid files. We find that invalid inputs are common in practice, about four percent (295 files) of all input files (7835 files) crawled from github were invalid.

Generic input repair with minimal data loss. We introduce the *ddmax* algorithm, automatically recovering a maximum of data from a given failure-inducing input (Section 5.3). To the best of our knowledge, *ddmax* is the first input repair technique that can be applied to arbitrary inputs and programs without additional knowledge on input formats or program code. In its evaluation on eight subjects and three input formats, using real-world invalid inputs as well as synthetic corruptions, we find that *ddmax* is effective: It repairs 69% of corrupted inputs and recovers about 78% of data, within a time budget of one minute per input.

An efficient syntactical input repair technique. We introduce a variant of *ddmax* that makes use of a *grammar* to parse inputs into derivation trees and to maximize inputs by pruning parts of the tree that could not be read (Section 5.4); this vastly speeds up input repair. In its evaluation, syntactic *ddmax* is faster and more efficient than the lexical variant.

Identifying faults in input data. The difference between the “repaired” input by *ddmax* and the original input contains all parts of the input that prevented the data from being processed in the first place. Section 5.5 shows that this difference precisely characterizes the fault in the input.

After discussing the threats to validity (Section 5.6) and limitations (Section 5.7), we discuss closely related work (Section 5.8). Lastly, we close with discussions and future work (Section 5.9).

5.2 Prevalence of Invalid Inputs

Before we start *repairing* inputs, let us first answer the question of how *relevant* the problem is. Is it actually possible that some application cannot open or process a data file? And would there be files claiming to adhere to some format if in fact, they are not? To answer such questions, we mine some invalid inputs from the wild.

Evaluation Setup

Subject Programs

In this study, we use eight programs as test subjects, namely **Blender** [221], **Assimp** [222], **Appleseed** [223], **JQ** [224], **JSON-Simple** [225], **Minimal-JSON** [226], **Graphviz** [227], and finally

Table 11: Subject Programs

Subject Program	Input Format	Prog. Lang.	Size (in KLoC)	Maturity (1 st Commit)
Blender	OBJ	C/C++	1800	Jan. 1994
Assimp	OBJ	C++	88.9	July 2002
Appleseed	OBJ	C++	600.1	May 2009
JQ	JSON	C	20.2	July 2012
JSONSimple	JSON	Java	2.6	Nov. 2008
Minimal-JSON	JSON	Java	6.4	Feb. 2013
Graphviz	DOT	C	1140	Sep. 1991
Gephi	DOT	Java	166.1	July 2008

Table 12: Input Grammar Details

Grammar	Size (LoC)	#ParserRules	#LexerRules
JSON	79	5	9
Wave. OBJ	271	13	42
DOT	181	14	15

Gephi [228]. Each input format was evaluated with three subjects, except for *DOT* which was evaluated with two programs. All our subject programs are open source **C**, **C++** or **Java** programs. On average, these programs have 478 KLoC and a maturity of over 14 years. Table 11 highlights the properties of our subject programs.

Grammars

We have collected the grammars for our subjects from the *ANTLR Grammar repository* [229]. We chose complex and large grammars for data-rich input formats used in two popular domains, namely graphics domain (i.e. **Wavefront OBJ** (OBJ) and DOT) and data exchange domain (i.e. JSON). To ensure the grammars were sound, we tested them with 50 valid crawled files for each input format. We modified the **Wavefront OBJ** grammar since its ANTLR grammar was only a subset of the official Wavefront OBJ specification [230]. The JSON and DOT grammars were used unmodified since they matched the official specifications [231, 232]. On average, the grammars are written in 177 LoC, with 11 parser rules and 22 lexer rules (*cf. Table 12*).

Mining and Filtering Input Files.

Table 13 highlights the details of the input files in our corpus. We crawled for a specific file format using the file extension (e.g. “.json” for the JSON input format). In total, we collected a corpus of 9544 input files (*cf. #Crawled Files in Table 13*) using the *Github API* for crawling [233]. Then, we deleted all files that are empty or duplicated, as well as the input files that have a different input format despite having the intended file name suffix (e.g. a Wavefront OBJ file has the same suffix (“.obj”) as a binary OBJ file that was created by a compiler). This resulted in 7835 unique input files (*cf. #Unique files*). We also separated files that contain unsupported grammar extensions. In particular, for JSON and DOT, we removed 166 input files (*cf. #Files Rejected by A*) that

Table 13: Details of Mined Input Files. We report the *cause of input invalidity* by showing the *number of files rejected by* (A) the *input grammar*, (B) *at least one subject program* and (C) *all subject programs*

Input Format	#Crawled Files	#Unique Files	#Valid Files	#Invalid Files	#Files Rejected by			Mean File Size (KiB)	
					A	B	C	Valid	Invalid
JSON	8654	7006	6948	222	164	58	52	12.84	0.78
OBJ	509	480	455	25	0	25	0	401.57	64.15
DOT	381	349	303	48	2	46	4	4.74	2.88

only contain literals like a number or a string (e.g. which are invalid JSON [231]) and JSON files that contain multiple JSON files appended to each other, as written by some programs.

To determine actual invalid input files (*cf.* *#Invalid*), we *filter* out the valid input files from the set of unique files by checking that (1) the file does not lead to a lexing/parsing error when parsed by *ANTLR* and (2) the file was successfully opened by all subject programs (of the input format) without crashing (using the test oracle in Section 5.2). In total, 7702 input files (*cf.* *#Valid Files*) passed the check of the filtering process and the remaining 295 input files represent our set of real-world invalid files (*cf.* *#Invalid Files*). Exactly 166 inputs were rejected by ANTLR, this is shown in Table 3 (*cf.* *#Files Rejected by A*).

In the set of *invalid input files*, 129 input files specifically failed to be processed by at least one of the subject programs (*cf.* *#Files Rejected by B*), while 56 input files failed for all subject programs (*cf.* *#Files Rejected by C*). In addition, we modified all valid files by removing additional white spaces in each, in order to accurately determine the data loss incurred by our approaches during evaluation.

Test Oracle

In our setup, the *test oracle* for *ddmax* is a *crashing oracle*. An input is treated as *invalid* if it crashes the subject program, or the result of the subject is empty, or the subject takes more than 10 seconds to process the input⁴⁵. A program run is considered a crash if the subject program returns a non-zero exit value. Even if a subject reports an error, it is only considered a crash if it also returns a non-zero exit value. Opening a *valid* file, however, produces a non-empty output after 10 seconds and does not crash the subject program. The test oracle does not use ANTLR as an invalidity criterium for (lexical) *ddmax*, because the goal is to repair an input with feedback from a subject program, without the knowledge of the input grammar. Although, syntactic *ddmax* employs ANTLR to build its initial AST, it does not obtain feedback from ANTLR during repair, i.e. when the AST is being modified.

To automate tests, we ensure that all subject programs have a full command-line interface (CLI) support or a Java/Python API. The test oracle was implemented in 890 LoC of Java and 412 LoC of Python code.

⁴⁵This execution time of 10 seconds was determined as a maximum opening time to successfully process all valid input files in our evaluation corpus.

Experimental Results

RQ1: How prevalent are invalid inputs in practice?

Invalid input files are common. About *four percent* of all inputs in our corpus (295 files) were invalid (*cf. Table 13*); they were either rejected by subject program(s) or the input grammar. Specifically, about two percent of the input files (129 files) in our sample were rejected by at least one subject program; however, less than 1% (56 files) were rejected by *all* subject programs in our evaluation setup.

A common cause of invalidity is wrong syntax, missing or non-conforming elements. Many input files were invalid because of single character errors, such as a deleted character, a missing character or an extraneous character. For instance, some JSON inputs were invalid due to deletions of characters such as quotes, parentheses and braces. These errors are difficult to find because they are often hidden in large documents. For example, our set of crawled OBJ files contained many files of about 300KiB with one corrupted line (e.g. an invalid character inside a “usemtl” statement). To fix such an error by hand, one would have to scroll through thousands of lines of code and find this single corrupted character. Other sources of invalidity include the addition of elements that do not conform with the input specification. Some JSON files contained comments that begin with the “\$” character. Comments are not permitted in JSON, however, this was common practice in some JSON files and a few parsers support comments (e.g. Google Gson).

In our sample of GitHub files, four percent could not be processed either by the input grammar or at least one subject program.

5.3 Lexical Repair

Now that we have established that there are actually files that cannot be properly parsed or opened, let us introduce the *ddmax* algorithm for recovering and repairing invalid input. This variant of *ddmax* works on a character-by-character basis; we thus call it *lexical ddmax*.

Delta Debugging

Our *ddmax* technique can be seen as a variation on *the minimizing delta debugging algorithm*, a technique for automatically reducing failure-inducing inputs by means of systematic tests. The *reduction problem* is modeled as follows: *Configurations* consisting of individual (input) elements which may or may not be present. There are two configurations: a *passing configuration* c_\checkmark and a *failing configuration* c_\times . The passing configuration c_\checkmark typically stands for an empty or trivial input ($c_\checkmark = \emptyset$), and the failing configuration $c_\times \supset c_\checkmark$ stands for the failure-inducing input in question. In our example from Section 5.1, the failing configuration would be

$$c_\times = \{ \text{"item": "Apple", "price": **3.45 } \} \quad (1)$$

Zeller et al. [57] define the *ddmin* algorithm as follows. *ddmin* produces one set c'_\times with $c_\checkmark \subset c'_\times \subseteq c_\times$, where c'_\times has a *minimal size overall*. It works by testing sets c' that lie *between* c_\checkmark

Maximizing Delta Debugging Algorithm

Let $test$ and $c_{\mathbf{x}}$ be given such that $test(\emptyset) = \checkmark \wedge test(c_{\mathbf{x}}) = \mathbf{x}$ hold. The goal is to find $c'_{\checkmark} = dmax(c_{\mathbf{x}})$ such that $c'_{\checkmark} \subset c_{\mathbf{x}}$, $test(c'_{\checkmark}) = \checkmark$, and $\Delta = c_{\mathbf{x}} - c'_{\checkmark}$ is 1-minimal.

The *maximizing Delta Debugging algorithm* $dmax(c)$ is

$$dmax(c_{\mathbf{x}}) = dmax_2(\emptyset, 2) \quad \text{where}$$

$$dmax_2(c'_{\checkmark}, n) = \begin{cases} dmax_2(c_{\mathbf{x}} - \Delta_i, 2) & \text{if } \exists i \in \{1, \dots, n\} \cdot test(c_{\mathbf{x}} - \Delta_i) = \checkmark \\ & \text{("increase to complement")} \\ dmax_2(c'_{\checkmark} \cup \Delta_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \dots, n\} \cdot test(c'_{\checkmark} \cup \Delta_i) = \checkmark \\ & \text{("increase to subset")} \\ dmax_2(c'_{\checkmark}, \min(|c_{\mathbf{x}}|, 2n)) & \text{else if } n < |c_{\mathbf{x}} - c'_{\checkmark}| \\ & \text{("increase granularity")} \\ c'_{\checkmark} & \text{otherwise ("done").} \end{cases}$$

where $\Delta = c_{\mathbf{x}} - c'_{\checkmark} = \Delta_1 \cup \Delta_2 \cup \dots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c_{\mathbf{x}} - c'_{\checkmark}|/n$ holds.

The recursion invariant (and thus precondition) for $dmax_2$ is $test(c'_{\checkmark}) = \checkmark \wedge n \leq |\Delta|$.

Figure 46: Maximizing Lexical Delta Debugging algorithm

and $c_{\mathbf{x}}$ (i.e., $c_{\checkmark} \subseteq c' \subseteq c_{\mathbf{x}}$). A test involves running the original program on the newly synthesized input c' . The outcome $test(c')$ of the test—either \checkmark (passing), \mathbf{x} (failing), or $?$ (unresolved)—determines algorithm progress: Whenever a subset $c' \subseteq c_{\mathbf{x}}$ fails ($test(c') = \mathbf{x}$), $ddmin$ further narrows down the difference between c' and c_{\checkmark} . In our example from Section 5.1, Figure 43 shows a typical $ddmin$ output $c'_{\mathbf{x}}$: The one character in the input suffices to cause the (syntax) error.

When choosing a new candidate c' , $ddmin$ initially splits the sets to be tested in half; as long as tests always pass or fail, this is as efficient as a binary search. If tests are unresolved (say, because the input is invalid), $ddmin$ resorts to cutting quarters, eighths, sixteenths of the input ($ddmin$). Eventually, $ddmin$ tests each remaining element (character) for its relevance in producing the failure.

The $dmax$ Algorithm

Our definition of $dmax$ is shown in Figure 46. $dmax$ uses the same setting as $ddmin$; however, rather than *minimizing* the failure-inducing input $c_{\mathbf{x}}$, it starts with a passing input $c'_{\checkmark} = c_{\checkmark}$; like $ddmin$, it assumes for simplicity that $c_{\checkmark} = \emptyset$ holds. It then *maximizes* c'_{\checkmark} , systematically minimizing the difference between c'_{\checkmark} and $c_{\mathbf{x}}$ using the same techniques as $ddmin$ (first progressing with large differences, then smaller and smaller differences), until every remaining difference would cause c'_{\checkmark} to fail. This makes $dmax$ act in exact symmetry to $ddmin$, and complements the original definitions of dd and $ddmin$ [57].

A *ddmax* Example

How does *ddmax* work? Let us illustrate it on the example from Section 5.1. We have $c_{\mathbf{x}}$ defined as in Equation (1), above, and evaluate $ddmax(c_{\mathbf{x}})$ to obtain $c'_{\mathbf{v}}$, the maximal subset of $c_{\mathbf{x}}$ that passes the test (i.e., that can be still be processed by our JSON application at hand). We now invoke $ddmax_2(c_{\mathbf{x}})$ and get $ddmax_2(\emptyset, 2)$ —that is, $c'_{\mathbf{v}} = \emptyset$ and $n = 2$. The set $c'_{\mathbf{v}}$ will continually hold more and more characters, and n will hold the current granularity.

$ddmax_2$ determines $\Delta = c_{\mathbf{x}} - c'_{\mathbf{v}} = c_{\mathbf{x}} - \emptyset = c_{\mathbf{x}}$, and splits it into two parts $\Delta_1 \cup \Delta_2 = \Delta$:

$$\begin{aligned}\Delta_1 &= \text{"price": **3.45 } \\ \Delta_2 &= \{ \text{"item": "Apple",}\end{aligned}$$

As part of “increase to complement”, $ddmax_2$ first tests $c_{\mathbf{x}} - \Delta_1$ (which is Δ_2) and then $c_{\mathbf{x}} - \Delta_2$ (which is Δ_1). Neither of both is a valid JSON input, hence the tests do not pass. In “increase to subset”, the sets to be tested are $c'_{\mathbf{v}} \cup \Delta_1 = (\emptyset \cup \Delta_1) = \Delta_1$ and $c'_{\mathbf{v}} \cup \Delta_2 = (\emptyset \cup \Delta_2) = \Delta_2$; we already know that these tests do not pass. Hence, we “increase granularity” and double n to $n = 4$.

With $n = 4$, we now split Δ into four parts $\Delta_1 \cup \dots \cup \Delta_4 = \Delta$:

$$\begin{aligned}\Delta_1 &= \{ \text{"item":} & \Delta_2 &= \text{"Apple",} \\ \Delta_3 &= \text{"price":} & \Delta_4 &= \text{**3.45 } \end{aligned}$$

In “increase to complement”, the tests run on the failing set $c_{\mathbf{x}}$ *without* the individual Δ_i —that is:

$$\begin{aligned}c_{\mathbf{x}} - \Delta_1 &= \text{"Apple", "price": **3.45 } \\ c_{\mathbf{x}} - \Delta_2 &= \{ \text{"item": "price": **3.45 } \\ c_{\mathbf{x}} - \Delta_3 &= \{ \text{"item": "Apple", **3.45 } \\ c_{\mathbf{x}} - \Delta_4 &= \{ \text{"item": "Apple", "price":}\end{aligned}$$

None of these inputs is syntactically valid JSON, and no test passes; so *ddmax* further increases granularity to $n = 8$. In this round, again none of the Δ_i pass; but one of the complements does:

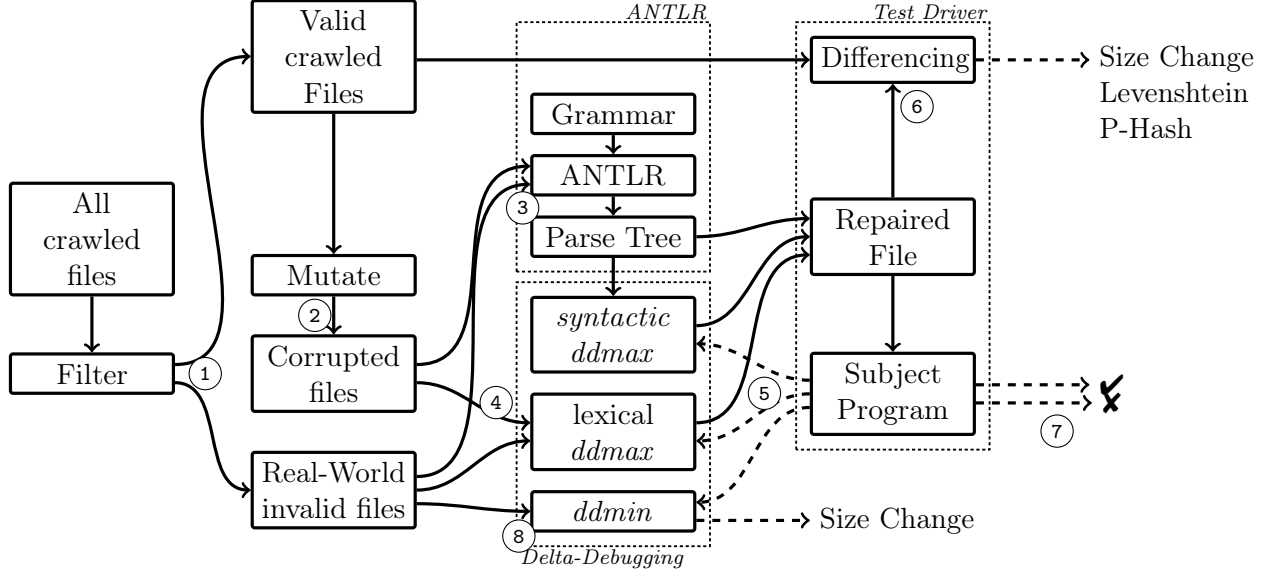
$$c_{\mathbf{x}} - \Delta_6 = \{ \text{"item": "Apple", "price":45 } \}$$

$$\text{with } \Delta_6 = \text{**3.}$$

The set $c_{\mathbf{x}} - \Delta_6$ is indeed a syntactically valid JSON input, and $test(c_{\mathbf{x}} - \Delta_6)$ passes (“increase to complement”). At this point, we have recovered $\frac{31}{36} = 86\%$ of the input data already.

Can we add more characters? Following the *ddmax* definition, we reinvoke $ddmax_2$ with $c'_{\mathbf{v}} = c_{\mathbf{x}} - \Delta_6$. Now, the remaining difference between $c'_{\mathbf{v}}$ and $c_{\mathbf{x}}$ is Δ_6 as above. We restart with $n = 2$ and decompose the remaining $\Delta = c_{\mathbf{x}} - c'_{\mathbf{v}} = \Delta_6$ into Δ_{6_1} and Δ_{6_2} :

$$\Delta_{6_1} = \text{**} \quad \Delta_{6_2} = \text{3.}$$

Figure 47: Workflow of the *ddmax* evaluation

Now, $c_{\mathbf{x}} - \Delta_{61}$ passes, yielding the syntactically correct input:

$$c_{\mathbf{x}} - \Delta_{61} = \{ \text{"item": "Apple", "price":3.45 } \}$$

A further iteration will also recover the space character before the number, eventually yielding the repaired input in Figure 44 and the remaining difference Δ in Figure 45.

The example demonstrates two important properties of *ddmax*:

- *ddmax* is *thorough*. Its result c'_{\checkmark} is *1-maximal*—that is, adding any further character from $c_{\mathbf{x}}$ will no longer pass. Formally, this means that $\forall \delta_i \in c_{\mathbf{x}} - c'_{\checkmark} \cdot \text{test}(c'_{\checkmark} \cup \{\delta_i\}) \neq \checkmark$ holds.⁴⁶
- *ddmax* can be *slow*. The *complexity* of *ddmax* is the same as *ddmin*—in the worst case, the number of tests carried out by $\text{ddmax}(c_{\mathbf{x}})$ is $|c_{\mathbf{x}}|^2 + 3|c_{\mathbf{x}}|$; and in the best case—if there is only one failure-inducing change $\Delta_i \in c_{\mathbf{x}}$, and all cases that do *not* include Δ_i pass, then the number of tests t is limited by $t \leq 2 \log_2(|c_{\mathbf{x}}|)$.

In practice, as with *ddmin*, things will be somewhere between the two extremes; but keep in mind that at maximum granularity, *ddmax* runs at least $|c_{\mathbf{x}} - c'_{\checkmark}|$ tests—that is, one test for every character that possibly still could be restored.

With these properties, what we get with *ddmax* is an algorithm that guarantees a maximum of data recovery, albeit at the price of possibly running a large number of tests.

Evaluation Setup

Workflow

Figure 47 shows the workflow of our evaluation. First, we collect real-world invalid input files from the set of *crawled files*, according to Section 5.2. Those files are then filtered into a set of valid files and a set of invalid files (Step 1) and duplicates and files with a wrong format are deleted.

⁴⁶Both maximality and complexity properties are proven in a way analogous to the properties of *ddmin* in [57].

Secondly, we select and mutate 50 valid *crawled files* to produce an additional set of corrupted input files (Step 2). Then, we feed a invalid file to each subject program, and the *ANTLR* parser framework (Step 3). *ANTLR* executes its default *error recovery strategy* while generating a parse tree for the input. Next, we feed the invalid file to *lexical ddmax* (Step 4). *Lexical ddmax* tests the input under repair repeatedly using the feedback from the subject program (Step 5). Then, we feed the original crawled files and the resulting repaired file from each technique to the *differencing* framework (Step 6), which computes the *change in file size*, *Levenshtein distance* and *perceptive hash value* for both files. We save the feedback from our subject program (Step 7). Finally, to ensure the quality of our approach, we also execute *ddmin* on the real-world invalid inputs (Step 8) and report the content and size of the result.

Lexical ddmax was implemented in 595 LoC of **Java** code. *ANTLR* also implements an inbuilt error recovery strategy which is designed to recover from lexing or parsing errors (e.g. missing/wrong tokens or incomplete parse trees) [234].

Mutations

In addition to the real-world invalid inputs (*cf. Section 5.2*), we also simulate real-world data corruption by applying *byte-level* mutations on valid input files. These mutations were chosen because they are similar to the corruptions observed in real-world invalid files (*see Section 5.2 and Section 5.5*). We perform the following mutations at a random position in each valid input file: *byte insertion*, *byte deletion* and *byte flip*. To simulate *single data corruption*, we randomly choose one of these mutations and apply it once on the valid input file. For *multiple data corruptions*, we perform up to 16 random mutations on each input file. A mutation is only successful (for an input format), if *at least* one of the subject programs (that passes before) fails after the mutation. These criteria is similar to how we collected invalid input files in the wild.

Metrics and Measures

In order to determine the quality of *ddmax* repair, we use the following metrics and tools:

1. **File Size:** We measure the *file size* of the inputs recovered by *ddmax* and the *difference in file size* between the original *valid input* and the *repaired file*. We use these measurements to account for the amount of data recovered by *ddmax* as well as the amount of data loss incurred.
2. **Levenshtein Distance:** Additionally, we measure data loss using the Levenshtein distance metric [235], measuring the *edit distance* between *valid input* and *repaired file*.
3. **Perceptive Image Difference:** In order to measure the (*semantic*) *information loss* incurred by *ddmax*, we calculate the hash value of our 3D images, i.e. **Wavefront OBJ** format. We compute the image distance of our 3D image files by rendering both the repaired image and the original valid image into several 2D images from three different camera angles and three scales, then measuring the 2D image distance of all nine images. We compare these images using the **Python ImageHash library** [236] in order to obtain a good approximation of the real image difference between those two 3D models as a

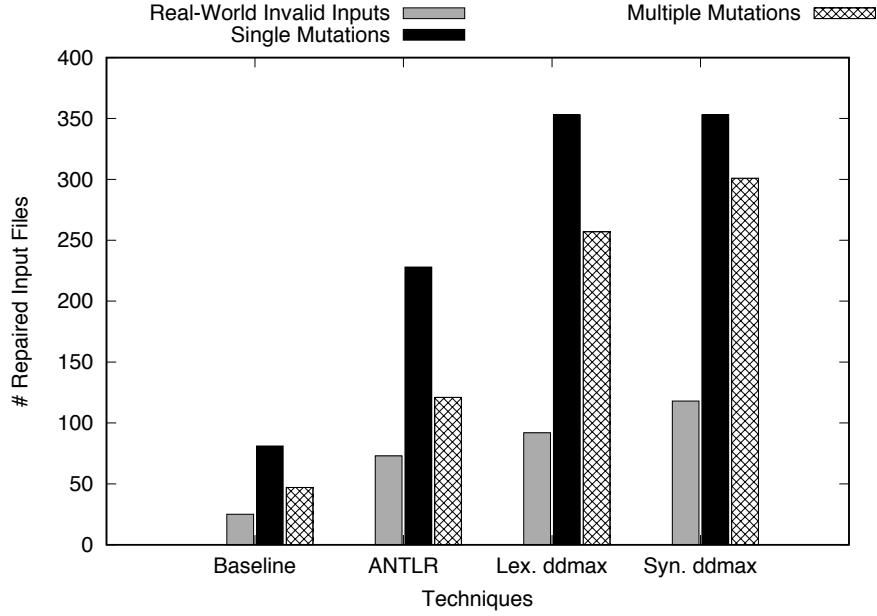


Figure 48: Number of Repaired Files for Each Technique

perceptive image difference between both images. In our setup, we use two rendering engines (Blender [221] and Appleseed [223]) to render the images.

Research Protocol

For each input format, we collect real-world invalid input files. Secondly, we perform single and multiple mutations on 50 valid input documents. Then, we execute all files on the different subject programs, in order to determine the number of input files which fail for each subject program. We proceed to run *lexical ddmax* on each invalid or mutated input file. In particular, we are interested in determining the following: (1.) **Baseline**: the number of invalid input files which are accepted by a subject program as *valid inputs* (i.e. non-failure-inducing inputs processed by the program without leading to a crash), in order to measure the *effectiveness* of the built-in *error recovery* feature of the program; and (2.) **ANTLR**: the number of invalid inputs which are repaired by ANTLR inbuilt *error recovery strategy*; (3) **Lexical**: the number of invalid inputs which are repaired by *lexical DDMax*.

All experiments were conducted on a Lenovo Thinkpad with four physical cores and 8GB of RAM, specifically an Intel(R) Core i7 2720qm @ 2.20GHz, 8 virtual cores, running 64-bit Arch Linux. All our prototypes are single-threaded.

Experimental Results

RQ2: How *effective* is *lexical ddmax* in repairing invalid input documents within a time budget of one minute per file?

Lexical ddmax repaired about two-thirds (66%) of all invalid inputs (*cf. Table 14*). It also outperformed both the in-built repair strategy of the subject programs (*Baseline*) and the ANTLR error recovery strategy (*ANTLR*), both of which repaired 14% and 40% of all invalid

Table 14: *ddmax* Effectiveness on All Invalid Inputs

Type of Invalidity	Format (#subjects)	#Possible Repairs	# repaired input files			
			Baseline	ANTLR	Lexical	Syntactic
Real World	JSON (3)	167	0	40	38	62
	OBJ (3)	33	1	8	24	25
	DOT (2)	64	24	25	30	31
Single Mutation	JSON (3)	150	4	80	115	127
	OBJ (3)	150	34	82	146	144
	DOT (2)	100	43	66	92	82
Multiple Mutation	JSON (3)	150	4	45	79	112
	OBJ (3)	150	3	29	127	126
	DOT (2)	100	40	47	51	63
Total (3)		1064	153	422	702	772

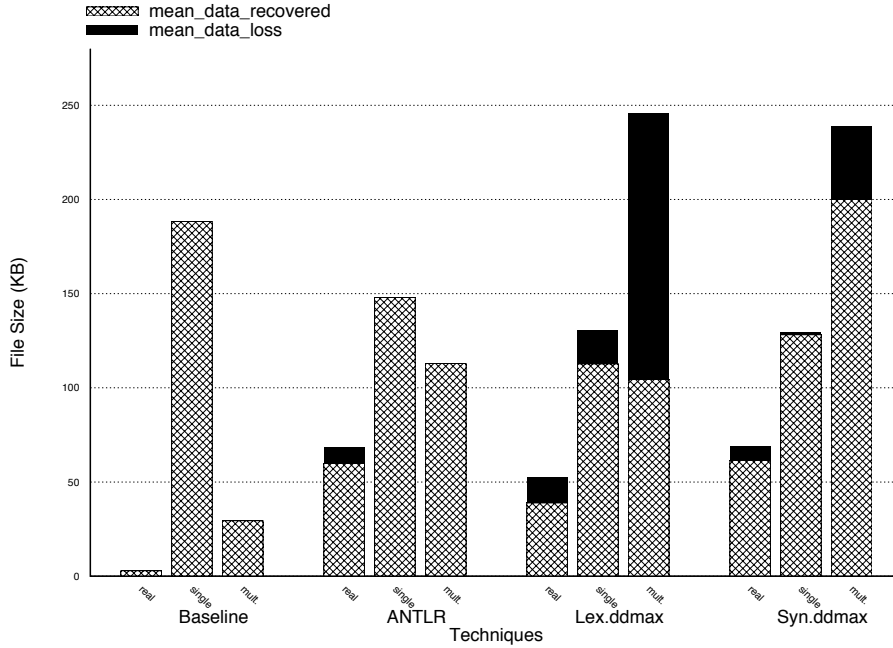


Figure 49: Data Recovered and Data Loss for all Inputs

input files respectively. Specifically, *lexical ddmax* repaired over four times as many invalid input files as the *Baseline* and 66% more invalid input files than ANTLR (*cf.* Figure 48). The performance of *lexical ddmax* was significantly better for both real-world invalid files and mutated invalid files.

Lexical ddmax repaired about two-thirds of all invalid inputs and significantly outperforms both the baseline and ANTLR.

RQ3: How much data is *recovered* by *lexical ddmax* and how much is the *data loss* incurred by *lexical ddmax*?

In terms of recovery rate, *lexical ddmax* performs slightly worse than the other techniques, with a recovery rate of 75% on real-world invalid inputs, 86% on single data corruption, and about 43%

Table 15: *ddmax* Efficiency on All Invalid Inputs for each technique (A) Baseline, (B) ANTLR, (C) Lexical *ddmax*, (D) Syntactic *ddmax*.

Type of Invalidity	Input Format	<i>Runtime (sec.)</i>				<i>#Runs</i>	
		A	B	C	D	C	D
Real World	JSON	2	2	1227	153	341525	6029
	OBJ	44	47	2065	1279	6253	3164
	DOT	48	166	3828	3018	2783	1162
Single Mutation	JSON	4	4	1584	1065	45651	129659
	OBJ	491	672	6151	4083	3809	1352
	DOT	58	60	1239	1244	6077	4565
Multiple Mutation	JSON	10	10	5903	2153	1194577	448801
	OBJ	624	728	9938	8132	8577	5043
	DOT	60	60	3365	2241	34876	11956
Mean		153	200	3981	2624	72296	70049

on multiple data corruption (*see Figure 49*). For both types of data invalidity, the *baseline* and *ANTLR* maintain an almost perfect data recovery rate (approximately 100%).

Lexical ddmax recovered most (75% and 65%) of the input data in real-world invalid inputs and mutated input files (, respectively).

In theory, *lexical ddmax* is guaranteed to ensure minimal data loss for all repairs. However, due to large file sizes and timeout constraints in our experimental setup, *lexical ddmax* often halts before the maximal valid data is recovered. In our experiment, *lexical ddmax* had timed out for 163 input files during repair. In order to inspect the data recovery rate of each approach in a more balanced setting, we examined the set of input files that were repaired by both *ANTLR* and *lexical ddmax*, before *lexical ddmax* timed out. In total, 109 repairs were accomplished by both *lexical ddmax* and *ANTLR*, before a time out. The data loss of *lexical ddmax* is minimal and comparable to *ANTLR*, this holds for both real-world and mutated invalid files for the intersecting set before timeout. In fact, on average, *lexical ddmax* recovered 1.724 KiB of data, and *ANTLR* recovered 1.548 KiB.

*Overall, lexical ddmax incurs minimal data loss during repair:
It recovers a similar amount of data from invalid input files, in comparison to ANTLR.*

RQ4: How efficient is *lexical ddmax* in repairing invalid input documents?

On average, it took less than two minutes (1.3 minutes) to repair a file (*cf. Figure 54*). In comparison, both the *Baseline* and *ANTLR* had an execution time of 3 and 4 seconds per input file respectively. This indicates that *lexical ddmax* is more time-consuming than both the *Baseline* and *ANTLR*. This is expected since *ddmax* requires multiple executions of the subject programs (as indicated in *lexical #Runs* in Table 15).

*Lexical ddmax is relatively fast in repairing an invalid input file:
It takes less than two minutes (78 seconds) on average.*

```
{ "item": "Apple", "price" 3.45 }
```

Figure 50: Failing JSON input with missing colon

```
{ "item": "Apple" }
```

Figure 51: Repaired JSON input by *ddmax*

5.4 Syntactic Repair

We have seen that *ddmax* is general, but also *slow*: If one wants to recover a maximum of data, it runs a single test for every candidate character that can be recovered. Is it possible to speed things up, possibly by leveraging information on the input format? To this end, we introduce the *syntactic ddmax* algorithm, which improves the performance of *ddmax* using the knowledge of the input grammar.

The key insight is to execute *ddmax* on the *parse tree* of the input, instead of the input characters. Here, we analyze the input at the syntactical level, rather than the lexical level. This improves the runtime and general performance of the *ddmax* algorithm. The main benefit of the approach is that it enables *ddmax* to reason at a more coarse-grained level by testing on the input structure. *Lexical ddmax* may take thousands of test runs, depending on the size of the input, in fact its number of runs is bound to the number of characters in the input. However, *syntactic ddmax* is bound to the *number of terminal nodes* in the parse tree, which is typically smaller than the number of characters in the input. Thus, *syntactic ddmax* can easily exclude corrupted parse tree nodes or subtrees during test runs. Additionally, the knowledge of the input structure ensures that the resulting recovered inputs are syntactically valid. This helps in the case of syntax errors, large corrupted input region(s) and multiple data corruptions on the input (structure).

Specifically, the syntactic *ddmax* algorithm takes as input a *parse tree* for the corrupted input file (see Figure 52) and obtains a pre-order list of terminals in the parse tree. For instance, consider the corrupted JSON input in Figure 50. Repairing this input using the *lexical ddmax* algorithm results in the JSON input in Figure 51, which would take over 100 test runs. Even for this small example, *syntactic ddmax* enhances the performance of *ddmax* with the input grammar, reducing the number of test runs of *ddmax* to nine (9) and improving performance by ten fold (10x).

To repair the input (cf. Figure 50), syntactic *ddmax* first parses the input into a parse tree⁴⁷, shown in Figure 52. Next, we run the *ddmax* algorithm on the parse tree, removing terminal nodes (instead of single characters) in each iteration of *ddmax*⁴⁸. We define $c_{\mathbf{x}}$ as our failing configuration, which contains the terminal nodes of the parse tree from Figure 52.

Let us run the *ddmax* algorithm on our example terminal nodes. We invoke $ddmax(c_{\mathbf{x}})$ which results in $ddmax_2(\emptyset, 2)$, so inside $ddmax_2$, we have $c'_{\mathbf{v}} = \emptyset$ and $n = 2$. At first, our Δ is split into

⁴⁷*ANTLR* is capable of generating a parse tree for corrupted input files, it summarizes syntactically wrong symbols or trees into error nodes (similar to Figure 52).

⁴⁸Removing only the error node in the parse tree does not necessarily result in a non-failure-inducing input.

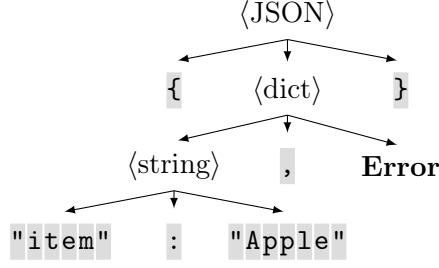


Figure 52: Parse tree of Figure 50

two parts⁴⁹:

$$\Delta_1 = \{ \text{"item"} : \text{"Apple"} \\ \Delta_2 = , \text{Error} \}$$

Running $\text{test}(c_{\mathbf{x}} - \Delta_1)$ and $\text{test}(c_{\mathbf{x}} - \Delta_2)$ both fail ($= \mathbf{x}$). We are at the first run, so with $c'_{\mathbf{x}} = \emptyset$, $c'_{\mathbf{x}} \cup \Delta_1 = c_{\mathbf{x}} - \Delta_2$ and $c'_{\mathbf{x}} \cup \Delta_2 = c_{\mathbf{x}} - \Delta_1$ which also both fail in the “increase to subset” step. Next, we set $n = 4$ and restart $\text{ddmax}_2(c'_{\mathbf{x}}, n)$.

With $n = 4$, in the “increase to complement” and “increase to subset” steps, we get

$$\begin{array}{ll} \Delta_1 = \{ \text{"item"} & \Delta_2 = : \text{"Apple"} \\ \Delta_3 = , \text{Error} & \Delta_4 = \} \end{array}$$

In the “increase to complement” step, we find that $\text{test}(c_{\mathbf{x}} - \Delta_3) = \checkmark$, so we repeat our algorithm with $c'_{\mathbf{x}} = c_{\mathbf{x}} - \Delta_3$ and $n = 2$, getting

$$\Delta_1 = , \quad \Delta_2 = \text{Error}$$

Since neither $\text{test}(c_{\mathbf{x}} - \Delta_i)$ nor $\text{test}(c'_{\mathbf{x}} \cup \Delta_i)$ passes for any i and $n = |c_{\mathbf{x}} - c'_{\mathbf{x}}| = 2$, we are done and end up with the remaining input seen in Figure 51. For this example, the *syntactic ddmax* run needed only 9 test runs of the subject program to repair the faulty input.

Let us now take a look at the complexity of our algorithm. As mentioned in Section 5.3, *ddmax* has a worst-case complexity of $t = |c_{\mathbf{x}}|^2 + 3|c_{\mathbf{x}}|$ test runs and a best-case complexity of $t \leq 2\log_2(|c_{\mathbf{x}}|)$. Intuitively, the complexity of *syntactic ddmax* is similar to the complexity of *ddmax*, except that it is bounded by the number of terminal nodes instead of the number of characters. In the worst case, an input’s parse tree has as many terminal nodes as characters. However, real-world input formats have keywords, data types and character classes to aggregate group of characters into terminals (e.g. string and integers). This reduces the number of terminal nodes and the required number of test runs for *syntactic ddmax*. It therefore speeds up *ddmax* by decreasing the number of elements to maximize with *ddmax*. Consider the example in Figure 50, there are 33 single characters to search with *lexical ddmax*, which are parsed into 7 terminal nodes for *syntactic ddmax*. In general, we can assume that with an average terminal node length

⁴⁹Note that checking for only syntactically valid subsets of the programs (e.g. using the grammar only) is not sufficient to repair the input. We leverage the application, since the semantics and intended use of the input file are encoded in the logic of the application.

of n characters, we have a worst-case complexity of $\left(\frac{|c_{\mathbf{x}}|}{n}\right)^2 + 3\frac{|c_{\mathbf{x}}|}{n}$ test runs and a best-case complexity of $t \leq 2\log_2\left(\frac{|c_{\mathbf{x}}|}{n}\right)$ test runs.

Evaluation Setup

Implementation

Syntactic ddmax was implemented in 1084 LoC of **Java** code, this implementation is built on top of the *ANTLR* 4.5 parser generator framework [237] for each input grammar. Overall, the implementation of *syntactic ddmax* differs from that of *lexical ddmax* in Section 5.3, because of its use of the input grammar and parse tree. Specifically, *Syntactic ddmax* uses the *ANTLR* parse tree (from Step 3 in Figure 47) to repair invalid inputs. In our evaluation, we feed the invalid real-world files into our *syntactic ddmax*, we proceed to run *syntactic ddmax* on each invalid input file and evaluate the change in file size (i.e. the data loss on byte-level). *Syntactic ddmax* tests the input under repair repeatedly using the feedback from the subject program (Step 5). In addition to the research protocol in Section 5.3, we feed all invalid input files to *syntactic ddmax* and measure the number of invalid files which are repaired by our *syntactic DDMax* using the input grammar, this measure is termed ***Syntactic***.

Experimental Results

RQ5: How effective is *syntactic ddmax* in repairing invalid input documents within a time budget of one minute per file?

Syntactic ddmax repaired about three-quarters (73%) of all invalid inputs in our evaluation, within a time budget of one minute per input (cf. Table 14). Overall, it is about 10% more effective than *lexical ddmax* (cf. Figure 48). It significantly outperformed both the built-in repair strategies of the subject programs and *ANTLR*, it repaired five times as many files as the subject programs, and almost twice as many files as *ANTLR* (cf. Table 14). This confirms our hypothesis (in **RQ2**) that *ddmax* can benefit from the knowledge of the input grammar.

Syntactic ddmax repaired about three-quarters of all invalid inputs (within one minute per input) and it is more effective than lexical ddmax, for all invalid inputs.

RQ6: How much data is recovered by *syntactic ddmax* and how much is the data loss incurred by *syntactic ddmax* ?

On average, *syntactic ddmax* (89%) has a higher data recovery rate in comparison to *lexical ddmax* (58%) for all invalid inputs. For single data corruption, the data recovery rate of *syntactic ddmax* is similar to that of *ANTLR* and the *baseline*, when using mean file size as a metric. On multiple data corruption, *syntactic ddmax* recovered about 84% of the data in the input files (cf. Figure 49). For all invalid inputs, the *baseline* and *ANTLR* maintain an almost perfect data recovery rate (approximately 100%). Evidently, the data loss incurred by both *ANTLR* and the *baseline* is negligible.

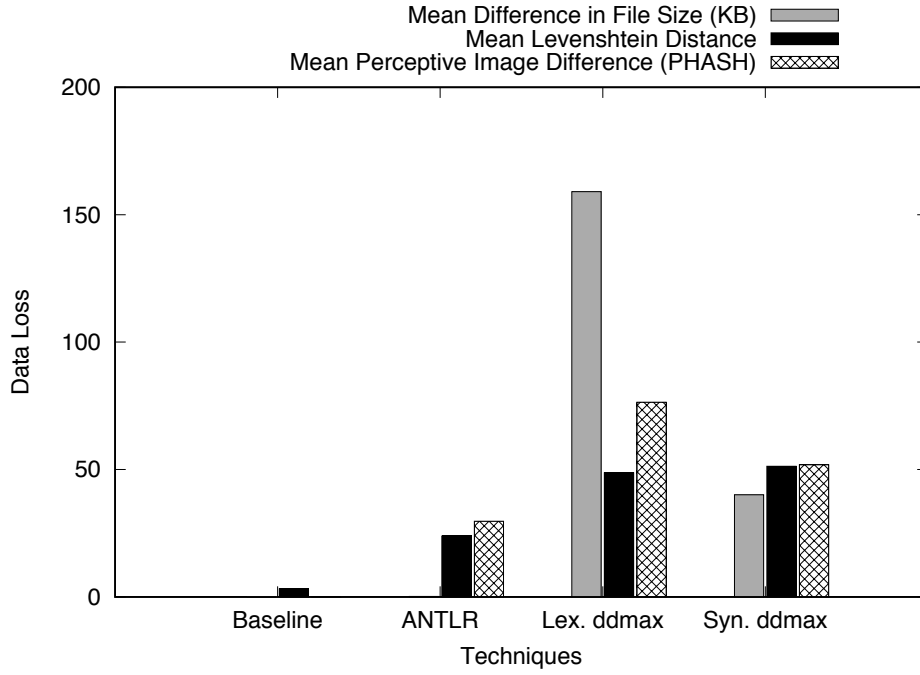


Figure 53: Data Loss Incurred for Invalid Files

Syntactic ddmax has a high data recovery rate, recovering most (89%) of the data in invalid input files.

The data loss incurred by *syntactic ddmax* is very low, in terms of the edit distance between the recovered file and the valid file. For all invalid inputs, it is less than 50% worse off than *ANTLR*, as captured by the *Levenshtein distance* (cf. Figure 53). In particular, the mean edit distance of the repaired file and the originally valid input file is less than four for the *baseline* and about 24 for *ANTLR*. As expected, the Levenshtein distance is lower (21–28) for single data corruptions for *lexical* and *syntactical ddmax* respectively, and higher for multiple corruptions (76–77). On inspection, we found that the high loss of *ddmax* is due to *early timeouts* for large input files, indeed, *ddmax* finds a valid subset, but times out before the maximal subset is reached. For Wavefront OBJ files, the perceptive image difference shows us similar scaling result as the Levenshtein distance. While it shows small results for *Baseline* and *ANTLR* (0.1 and 29.7, respectively), the results for *lexical* and *syntactical ddmax* are higher (76.4 and 51.9), thus the difference between the unmodified image and the repaired image is larger.

We conduct our evaluation of minimal data loss similarly to the setting in **RQ3** (cf. Section 5.3). As expected, *syntactic ddmax* recovered slightly less data than *lexical ddmax*, exactly 1.720 KiB on average. This is because *syntactic ddmax* removes terminal nodes, a terminal node may contain more characters than the number of mutated characters in the node. In summary, with a high enough timeout *lexical ddmax* is guaranteed to achieve minimal data loss, this guarantee does not hold for *syntactic ddmax*, since it operates at the parse tree level rather than the byte level.

Syntactic ddmax incurs comparatively similar data loss during repair as lexical ddmax.

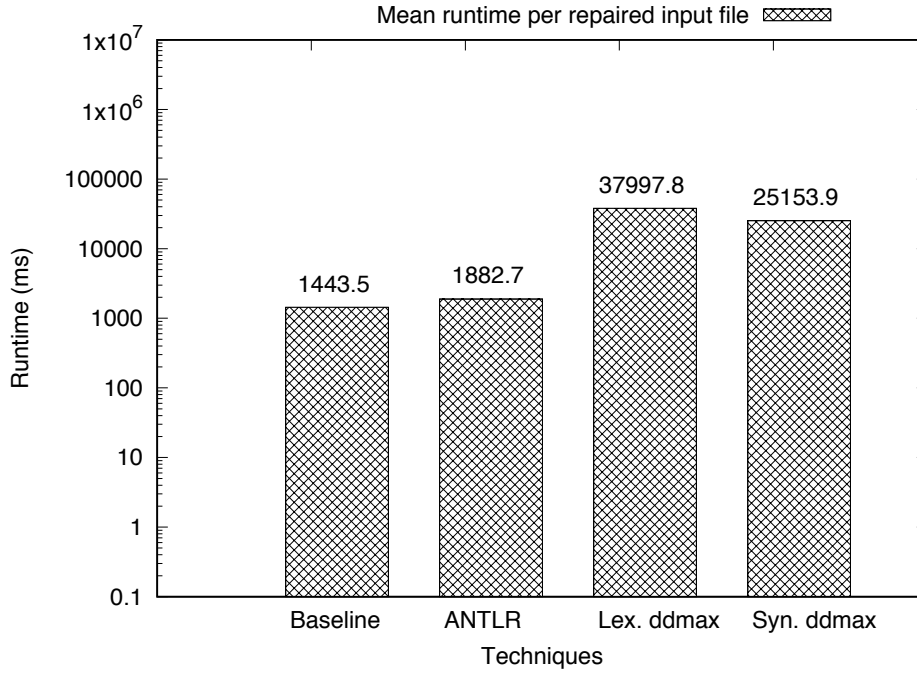


Figure 54: Mean Runtime per Input File for Each Technique

RQ7: How *efficient* is *syntactic ddmax* in repairing invalid input documents?

Syntactic ddmax improves over the runtime performance of *lexical ddmax* (cf. **RQ4** Section 5.3). It improves over *lexical ddmax* by 34%, its execution time is about two-third of the running time of *lexical ddmax*. Specifically, *syntactic ddmax* is quicker, it took approximately one minute to repair a single file, but requires a grammar and a parse tree⁵⁰.

*Syntactic ddmax is faster in repairing an invalid input file:
It takes less than one minute to repair a file on average.*

5.5 Diagnostic Quality

Even though *ddmax* is primarily meant for repairing data, its maximized input can also be useful for diagnostics and debugging. In particular, *ddmax* diagnosis is the difference (Δ) between the failing and maximal passing input. This is a minimal failure cause, which is necessary to debug the input. Most notably, the Δ from *ddmax* includes *all* input characters that are failure-inducing, whereas *ddmin* include only a minimal subset.

Evaluation Setup

To evaluate the diagnostic utility of *ddmax*, we compare *ddmax* diagnoses to the established state of the art input diagnosis approach *ddmin*. In our evaluation, we compare *ddmax* diagnosis to that of *ddmin*, we do not compare to the general delta debugging (DD) algorithm. This is because DD is not suited for repairing inputs. Although, DD would produce a passing and a failing input with a minimal difference between them. This DD difference could be as small as

⁵⁰Depending on the grammar and on the input file size, generating a parse tree should take less than a second.

Table 16: Diagnostic Quality on Real-World Invalid Inputs for (A) *ddmin* and (B) *ddmax* diagnoses, and (C) *ddmax* repair.

Format (#inputs)	Diagnosis (B)		Repair (B) (C)	Intersection (%)	
	(A)	(B)		(A) \cap (B)	(A) \cap (C)
JSON (21)	2.909	19.095	103.476	13.88	23.18
OBJ (18)	2.722	1.000	189.000	18.03	11.46
DOT (27)	376.654	1.115	675.346	5.76	54.64
Mean	155.754	6.804	360.747	11.69	32.85

the *ddmax* difference between the maximal passing input and the original failing input, and have similar diagnostic quality; also, DD would likely be faster. However, DD does not have the goal of minimizing data loss, and thus the passing input resulting from general DD may actually be close to a minimal input cutting away all the original context.

By construction, DD (and *ddmin*) can minimize (and thus lose) all the context of the original failure. For instance, if there is a flag in the input that activates the faulty function, and DD (and *ddmin*) will remove that flag, causing the program to pass, then this single flag will end up as failure-inducing input. On the other hand, *ddmax* would preserve as much of the original context as possible by construction. It is these experiences that have driven us to experiment with DD alternatives such as *ddmax* and *ddmin*.

We implemented a *ddmin* algorithm following the delta debugging algorithm in [57] in 450 LoC of Java code. Our *ddmin* implementation uses both the subject program and ANTLR as oracles to minimize an invalid input, in order to ensure that *ddmin* diagnosis is syntactically valid. This ensures that *ddmin* does not report a valid subset that may trigger a failure due to syntactic invalidity (e.g. in *cf. Figure 43 in Section 5.1*), since ANTLR can parse the *ddmin* diagnosis, but the subject program crashes. Then, we feed the real-world invalid files into our *ddmin* implementation (as seen in Figure 47 Step 8) and compare the diagnosis generated by *ddmin* to that of *ddmax*.

We are interested in evaluating the soundness and completeness of *ddmin* diagnosis, using *ddmax* diagnoses as the “ground truth”. To be fair to both approaches, we consider the intersection of the diagnoses for both *ddmin* and *ddmax* that finished execution before a time-out, this a set of 66 input files in total (*cf. Table 16*).

Experimental Results

RQ8: How effective is *ddmax* in diagnosing the root cause of invalid inputs, especially in comparison to *ddmin*?

Given that *ddmax* was completely executed without a timeout, the repair of *ddmax* is the *maximal passing input* and *ddmax* diagnosis is the *minimal failure cause*. As expected *ddmin* diagnosis is significantly larger (21 times more) than the *ddmax* diagnosis, hence, it contains a significant amount (33%) of the maximal passing input, which is considered noise in the diagnosis (*cf. (A) \cap (C) in Table 16*). Additionally, *ddmin* diagnosis only contains a small portion (12%) of the minimal failure cause required to diagnose the input invalidity (*cf. (A) \cap (B) in Table 16*). This

result shows that *ddmax* diagnosis is more effective for input debugging in comparison to the state of the art, *ddmin*.

*Only one-eighth (12%) of a *ddmin* diagnosis contains the minimal failure cause and about one-third (33%) of *ddmin* diagnosis contains the maximal passing input, on average.*

5.6 Threats to Validity

Our evaluation is limited to the following threats to validity:

External validity refers to the generalizability of our approach and results. We have evaluated our approach on a small set of applications and input grammars. There is a threat that *ddmax* does not generalize to other applications and grammars. However, we have mitigated this threat by evaluating *ddmax* using mature subject programs with varying input sizes. Our subjects have 478 KLoC and 14 years maturity, on average, making us confident that our approach will work on a large variety of applications and invalid inputs.

Internal validity is threatened by incorrectness of our implementation, specifically whether we have correctly adapted *ddmin* to *ddmax* for input repair. We mitigate this threat by testing our implementation on smaller inputs and simpler grammars, in order to ensure our implementation works as expected.

Construct validity is threatened by our test oracle, and consequently the error-handling implementation of the subject. For instance, an application which silently handles exceptions would not provide *ddmax* with useful feedback during test runs. We checked that the rendered input produced by the subject is non-empty, after a 10 second timeout, which was sufficient to identify failure-inducing inputs.

5.7 Limitations

Both *ddmax* variants are limited in the following ways:

Repair to subsets only. Both *ddmax* variants will return a strict lexical or syntactical *subset* of the original failure-inducing input. The assumption is that only data should be restored that already is there (rather than synthesizing new data, for instance). If the input format has several context-sensitive dependencies, such as checksums, hashes, encryption, or references, a strict lexical or syntactical subset might be small to the point of being useless.

Data repair, not information repair. Both *ddmax* variants are set to recover as much *data* as possible, but not necessarily *information*. Even though the repaired input may be lexically or syntactically close to the (presumed) original input, it can have very different semantics. Users therefore are advised to thoroughly *check the repaired input* for inconsistencies; the goal of this work is to *enable* users to load the input into their program such that they can engage in this task.

Input Semantics. Although, *ddmax* obtains some “semantic” information from the feedback of the subject program itself, this feedback is limited to failure characteristics, i.e. “pass” or “fail”. However, it is possible to extend *ddmax* to include (domain-specific) semantic checks, which could either be defined as the execution of specific program artifacts such as a specific branch, or programmatically defined by a developer (e.g. as an expected program output).

Multiple errors and multiple repairs. If there are multiple errors in an input, *ddmax* will produce a maximum input that repairs all of them. However, if there are multiple ways to repair the input, *ddmax* will produce only one of them. This property is shared with delta debugging and its variants, which also pick a local minimum rather than searching for a global one. However, it would be easy to modify *ddmax* to assess all alternative repairs rather than the first repair.

5.8 Related Work

There is a large body of work concerning the interplay of program, inputs, and faults. We discuss the most important related works.

Document Recovery attempts to fix broken input documents. Doccovery [63] uses symbolic execution to manipulate corrupted input documents in a manner that forces the program to follow an alternative error-free path. In contrast to *ddmax*, this is a white-box approach that analyzes the program paths executed by the failure-inducing inputs. S-DAGS [64] is a semi-automatic technique that enforces formal (semantic) consistency constraints on inputs documents in a collaborative document editing scenario. Both of these approaches require program analysis.

Input Rectification aims at transforming invalid inputs into inputs that behave acceptably. Input *rectifiers* [61, 62] address this problem by learning a set of constraints from typical inputs, then transforming a malicious input into a benign input that satisfies the learned constraints. In contrast, *ddmax* does not learn constraints but rather employs the feedback from the program’s execution (and a grammar) to determine an acceptable subset of the input. In comparison to security-critical rectification, its goal is maximal data recovery.

Input Minimization Numerous researchers have examined the problem of simplifying failure-inducing inputs [57, 58, 59, 60]. In particular, [59] (HDD) and [60] are closely related to *ddmax*. Both approaches use the input structure to simplify inputs, albeit without an input grammar. Unlike *ddmax*, these approaches do not recover maximal valid data from the failure-inducing input, but rather minimize the input like *ddmin*.

Data Diversity [65] transforms an invalid input into a valid input that generates an equivalent result, in order to improve *software reliability*. This is achieved by finding the regions of the input space that causes a fault, and re-expressing a failing input to avoid the faulty input regions. In contrast, *ddmax* does not require program analysis; it only needs a means to assess whether the program can process the input or not.

Data Structure Repair iteratively fixes corrupted data structures by enforcing that they conform to consistency constraints [238, 239, 240, 241]. These constraints can be extracted, specified and enforced with predicates [242], model-based systems [238], goal-directed reasoning [239], dynamic symbolic execution [240] or invariants [241]. On the one hand, the goal of data structure repair is to ensure a program executes safely and acceptably, despite data structure corruption. On the other hand, the goal of *ddmax* is to repair the input in order to avoid the corruption of the program’s internal data structure.

Syntactic Error Recovery. Parsers and compilers implement numerous syntax error recovery schemes [219, 243]. Most approaches involve a plethora of operations including insertion, deletion and replacement of symbols [244, 245, 246], extending forward or backwards from a parser error [247, 248], or more general methods of recovery and diagnosis [249, 250]. Unlike *ddmax*, these schemes ensure the compiler does not halt while parsing, however, the invalid input is not automatically fixed.

Data Cleaning and Repair. Several researchers have addressed the problem of data cleaning of database systems. Most approaches automatically analyse the database to remove noisy data or fill in missing data [251, 252]. Other approaches allow developers to write and apply logical rules on the database [253, 254, 255, 256, 257, 258]. In contrast to *ddmax*, all of these approaches repair database systems, not raw user inputs.

Data Testing and Debugging aims to identify system errors caused by well-formed but incorrect data while a user modifies a database [259]. For instance, continuous data testing (CDT) [260] identifies likely data errors by continuously executing domain-specific test queries, in order to warn users of test failures. DATAxRAY [261] also investigates the underlying conditions that cause data bugs, it reveals hidden connections and common properties among data errors. In contrast to *ddmax*, these approaches aim to guard data from new errors by detecting data errors in database systems during modification.

5.9 Discussions and Future Work

This chapter has provided empirical evidence on the relevance, causes and prevalence of invalid inputs in software practice. For instance, we found that four (4) percent of inputs in the wild are invalid. Evidently, debugging and repairing invalid inputs is a relevant and challenging problem. Thus, with *ddmax*, we have presented the first *generic* technique for automatically repairing failure-inducing inputs—that is, recovering a maximal subset of the input that can still be processed by the program at hand. Our approach (i.e. *ddmax*) is a variant of delta debugging that maximizes the passing input, both at a lexical and a syntactical level; it requires nothing more than the ability to automatically run the program with a given input. In our evaluation, we find that *ddmax* fully repairs 79% of invalid input files, within a one (1) minute time budget. Both variants of *ddmax* can be easily implemented and deployed in a large variety of contexts as they do not require any kind of program analysis.

This work opens the door for a number of exciting research opportunities, our future work will focus on the following issues:

Synthesizing input structures. Going for a strict lexical or syntactical subset of the failure-inducing input is a conservative strategy; yet, there can be cases where *adding* a small amount of lexical or syntactical elements can help to recover even more information. We are investigating appropriate grammar-based production strategies as well as hybrid strategies that leverage both syntactical and lexical progression.

Learned grammars. Right now, our syntactical variant of *ddmax* requires an input grammar to start with. We are investigating whether such a grammar can also be *inferred* from the program at hand [262, 263], thus freeing users or developers from having to provide a grammar.

From input repair to code repair. A minimal difference (Δ) between a maximized passing and a full failure-inducing input also brings great opportunities for fault localization and repair. For instance, what is the code executed by the failure-inducing input, but not by the maximized passing input? What are the differences in variable values? Such differences in execution can help developers to further narrow down failure causes as well as synthesizing code repairs.

End-user debugging. Our input repair technique needs no specific knowledge on program code, and could thus also be applied by end users. We are investigating appropriate strategies to communicate the results of our repair and information about conflicting document parts to end users, such that they can *fix the problem* without having to *fix the program*.

Hybrid repair. Lexical and syntactic *ddmax* can be combined such that after syntactic *ddmax* is executed on the parse tree, lexical *ddmax* further repairs the text in the faulty nodes. This combination reduces the number of iterations and the execution time, in comparison to lexical *ddmax*. Moreover, it improves on the effectiveness of syntactic *ddmax*.

Semantic Input Repair. It is possible to extend the *ddmax* test oracle to include checks for desirable “semantic” properties other than failure characteristics (i.e. pass or fail). For instance, the test oracle can be extended to check if some function is triggered or some specific output is produced, such “semantic” checks would ensure that the resulting maximized passing input is semantically similar to the original input and avoids the failure.

Fuzzing. Both variants of *ddmax* can be applied to improve software fuzzing. For instance, mutational fuzzing techniques often modify seed inputs to find bugs in the program. Often, these inputs become malformed after mutation, *ddmax* can be applied to repair such inputs, in order to ensure that they are valid, and consequently, cover program logic.

The implementation of *ddmax* and the experimental data are available as a replication package:

<https://tinyurl.com/debugging-inputs-icse-2020>

Learning Input Distributions for Grammar-Based Test Generation

This chapter is taken, directly or with minor modifications, from our 2020 TSE paper *Inputs from Hell: Learning Input Distributions for Grammar-Based Test Generation* [20]. My contribution in this work is as follows: (I) original idea; (II) partial implementation; (III) evaluation.

“A pinch of probability is worth a pound of perhaps.”

— James Thurber

6.1 Introduction

During the process of software testing, software engineers typically attempt to satisfy three goals:

1. First, the software should work well on *common* inputs, such that the software delivers its promise on the vast majority of cases that will be seen in typical operation. To cover such behavior, one typically has a set of dedicated tests (manually written or generated).
2. Second, the software should work well on *uncommon* inputs. The rationale for this is that such inputs would exercise code that is less frequently used in production, possibly less tested, and possibly less understood [70].
3. Third, the software should work well on inputs that *previously caused failures*, such that it is clear that previous bugs have been fixed. Again, these would be covered via specific tests.

How can engineers obtain such inputs? In this chapter, we propose an approach that *learns the distribution of input elements from typical inputs found in software practice, in order to drive test generation*. Firstly, our approach *learns the distribution of input elements from a set of sample inputs* found in software practice. The *learned input distribution is then used to drive the generation of new test inputs*. Specifically, we introduce a novel test generation method that produce additional inputs that are markedly *similar* or *dissimilar* to the sample. By learning from past failure-inducing inputs, we can create inputs with similar features; by learning from common inputs, we can create uncommon inputs with dissimilar features not seen in the sample.

The key ingredient to our approach is a *context-free grammar* that describes the input language to a program. Using such a grammar, we can *parse* existing input samples and *count* how frequently specific elements occur in these samples. Armed with these numbers, we can enrich the grammar to become a *probabilistic grammar*, in which production alternatives carry different likelihoods. Since these probabilities come from the samples used for the quantification, such a grammar captures properties of these samples, and producing from such a grammar should produce inputs that are similar to the sample. Furthermore, we can *invert* the learned probabilities in order to obtain a second probabilistic grammar, whose production would produce inputs that are *dissimilar* to the sample. We thus can produce three kinds of inputs, covering the three testing goals listed above:

1. **“Common inputs”**. By learning from *common* samples, we obtain a “common” probability distribution, which allows us to produce more “common” inputs. This is useful for regression testing.
2. **“Uncommon inputs”**. Learning from common samples, the resulting *inverted* grammar describes in turn the distribution of valid, but *uncommon* inputs. This is useful for completing test suites by testing uncommon features.
3. **“Failure-inducing inputs”**. By learning from samples that caused failures in the past, we can produce similar inputs that test the *surroundings* of the original inputs. This is useful for testing the completeness of fixes.

Both the “uncommon inputs” and “failure-inducing inputs” strategies have high chances of triggering failures. Since they combine features rarely seen or having caused issues in the past, we gave them the nickname “inputs from hell”. As an example, consider the following JavaScript input generated by focusing on uncommon features:

```
var { a: {} = 'b' } = {};
```

This snippet is valid JavaScript code, but causes the Mozilla Rhino 1.7.7.2 JavaScript engine to crash during interpretation.⁵¹ This input makes use of so-called *destructuring assignments*: In JavaScript, one can have several variables on the left hand side of an assignment or initialization. In such a case, each gets assigned a part of the structure on the right hand side, as in

```
var [one, two, three] = [1, 2, 3];
```

where the variable `one` is assigned a value of 1, `two` a value of 2, and so on. Such destructuring assignments, although useful in some contexts, are rarely found in JavaScript programs and tests. It is thus precisely the aim of our approach to generate such uncommon “inputs from hell”.

This chapter makes the following contributions:

1. We use context-free grammars to determine production probabilities from a given set of input samples.

⁵¹We have reported this snippet as Rhino issue #385 and it has been fixed by the developers.

2. We use mined probabilities to produce inputs that are *similar to a set of given samples*. This is useful for thoroughly testing commonly used features (regression testing), or to test the surroundings of previously failure-inducing inputs. Our approach thus leverages probabilistic grammars for both mining and test case generation. In our evaluation using the JSON, CSS and JavaScript formats, we show that our approach repeatedly covers the same code as the original sample inputs; learning from failure-inducing samples, we produce the same exceptions as the samples as well as new exceptions.
3. We use mined probabilities to produce inputs that are *markedly dissimilar* to a set of given samples, yet still valid according to the grammar. This is useful for robustness testing, as well as for exploring program behavior not triggered by the sample inputs. We are not aware of any other technique that achieves this objective. In our evaluation using the same subjects, we show that our approach is successful in repeatedly covering code not covered in the original samples.

The remainder of this chapter is organized as follows. After giving a motivational example in Section 6.2, we detail our approach in Section 6.3. Section 6.4 evaluates our three strategies (“common inputs”, “uncommon inputs”, and “failure-inducing inputs”) on various subjects. We then discuss the threats to validity and limitations in Section 6.5 and Section 6.6. Finally, we conclude by discussing related work and future work in Section 6.7 and Section 6.8, respectively.

6.2 Overview

To demonstrate how we produce both common and uncommon inputs, let us illustrate our approach using a simple example grammar. Let us assume we have a program P that processes *arithmetic expressions*; its inputs follow the standard syntax given by the grammar G below.

$$\begin{aligned}
 \text{Expr} &\rightarrow \text{Term} \mid \text{Expr} \text{ "+" } \text{Term} \mid \text{Expr} \text{ "-" } \text{Term}; \\
 \text{Term} &\rightarrow \text{Factor} \mid \text{Term} \text{ "*" } \text{Factor} \mid \text{Term} \text{ "/" } \text{Factor}; \\
 \text{Factor} &\rightarrow \text{Int} \mid \text{"+" } \text{Factor} \mid \text{"-" } \text{Factor} \mid \text{"(" Expr ")}; \\
 \text{Int} &\rightarrow \text{Digit Int} \mid \text{Digit}; \\
 \text{Digit} &\rightarrow \text{"0"} \mid \text{"1"} \mid \text{"2"} \mid \text{"3"} \mid \dots \mid \text{"9"};
 \end{aligned}$$

Let us further assume we have discovered a bug in P : The input $I = 1 * (2 + 3)$ is not evaluated properly. We have fixed the bug in P , but want to ensure that similar inputs would also be handled in a proper manner.

To obtain inputs that are *similar* to I , we first use the grammar G to *parse* I and determine the *distribution* of the individual choices in productions. This makes G a *probabilistic* grammar G_p in which the productions’ choices are tagged with their probabilities. For the input I above, for instance, we obtain the probabilistic rule

$$\begin{aligned}
 \text{Digit} &\rightarrow 0\% \text{"0"} \mid 33.3\% \text{"1"} \mid 33.3\% \text{"2"} \mid 33.3\% \text{"3"} \mid 0\% \text{"4"} \mid 0\% \text{"5"} \\
 &\quad \mid 0\% \text{"6"} \mid 0\% \text{"7"} \mid 0\% \text{"8"} \mid 0\% \text{"9"};
 \end{aligned}$$

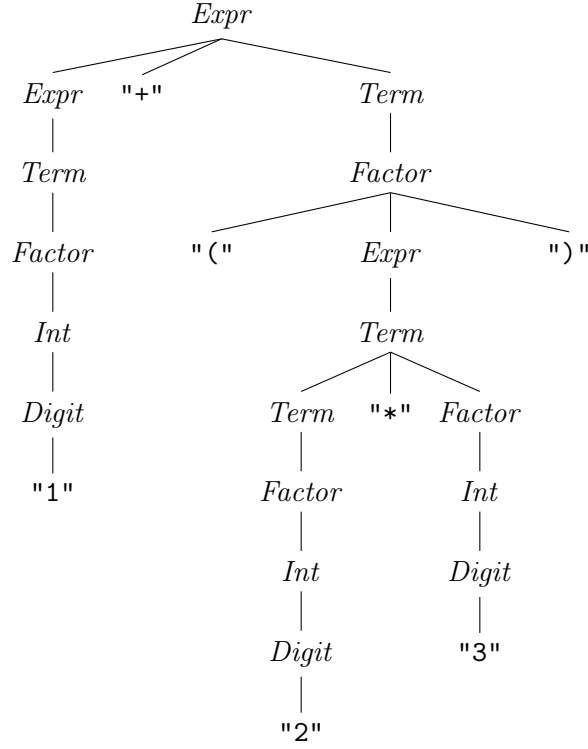


Figure 55: Derivation tree representing "1 + (2 * 3)"

$Expr \rightarrow 66.7\% Term \mid 33.3\% Expr \text{ "+" } Term \mid 0\% Expr \text{ "-" } Term;$
 $Term \rightarrow 75\% Factor \mid 25\% Term \text{ "*" } Factor \mid 0\% Term \text{ "/" } Factor;$
 $Factor \rightarrow 75\% Int \mid 0\% \text{ "+" } Factor \mid 0\% \text{ "-" } Factor \mid 25\% "(" Expr ")";$
 $Int \rightarrow 0\% Digit \mid 100\% Digit;$
 $Digit \rightarrow 0\% \text{"0"} \mid 33.3\% \text{"1"} \mid 33.3\% \text{"2"} \mid 33.3\% \text{"3"} \mid 0\% \text{"4"} \mid 0\% \text{"5"} \mid 0\% \text{"6"} \mid 0\% \text{"7"} \mid 0\% \text{"8"} \mid 0\% \text{"9"};$

Figure 56: Probabilistic grammar G_p , expanding G

(2 * 3)
 2 + 2 + 1 * (1) + 2
 ((3 * 3))
 3 * (((3 + 3 + 3) * (2 * 3 + 3))) * (3)
 3 * 1 * 3
 ((3) + 2 + 2 * 1) * (1)
 1
 ((2)) + 3

Figure 57: Inputs generated from G_p in Figure 56

```

Expr → 0% Term | 0% Expr "+" Term | 100% Expr "-" Term;
Term → 0% Factor | 0% Term "*" Factor | 100% Term "/" Factor;
Factor → 0% Int | 50% "+" Factor | 50% "-" Factor | 0% "(" Expr ")";
Int → 100% Digit Int | 0% Digit;
Digit → 14.3% "0" | 0% "1" | 0% "2" | 0% "3" | 14.3% "4" | 14.3% "5" | 14.3% "6"
        | 14.3% "7" | 14.3% "8" | 14.3% "9";

```

Figure 58: Grammar G_{p-1} inverted from G_p in Figure 56

which indicates the distribution of digits in I . Using this rule for production, we would obtain ones, twos, and threes at equal probabilities, but none of the other digits. Figure 56 shows the grammar G_p as extension of G with all probabilities as extracted from the derivation tree of I (Figure 55). In this derivation tree we see, for instance, that the nonterminal *Factor* occurs 4 times in total. 75% of the time it produces integers (*Int*), while in the remaining 25%, it produces a parenthesis expression ("*Expr*"). Expressions using unary operators like "+" *Factor* and "-" *Factor* do not occur.

If we use G_p from Figure 56 as a probabilistic production grammar, we obtain inputs according to these probabilities. As listed in Figure 57, these inputs uniquely consist of the digits and operators seen in our sample $1 * (2 + 3)$. All of these inputs are likely to cover the same code in P as the original sample input, yet with different input structures that trigger the same functionality in P in several new ways.

When would one want to replicate the features of sample inputs? In the “common inputs” strategy, one would create test cases that are similar to a set of common inputs; this is helpful for regression testing. In the more interesting “failure-inducing inputs” strategy, one would learn from a set of failure-inducing samples to replicate their features; this is useful for testing the surroundings of past bugs.

If one only has sample inputs that work just fine, one would typically be interested in inputs that are *different* from our samples—the “uncommon inputs” strategy. We can easily obtain such inputs by *inverting* the mined probabilities: if a rule previously had a weight of p , we now assign it a weight of $1/p$, normalized across all production alternatives. For our *Digit* rule, this gives the digits not seen so far a weight of $1/0 = \infty$, which is still distributed equally across all seven alternatives, yielding individual probabilities of $1/7 = 14.3\%$. Proportionally, the weights for the digits already seen in I are infinitely small, yielding a probability of effectively zero. The “inverted” rule reads now:

```

Digit → 14.3% "0" | 0% "1" | 0% "2" | 0% "3" | 14.3% "4" | 14.3% "5" | 14.3% "6"
        | 14.3% "7" | 14.3% "8" | 14.3% "9";

```

Applying this inversion to rules with non-terminal symbols is equally straightforward. The resulting probabilistic grammar G_{p-1} is given in Figure 58.

This inversion can lead to infinite derivations, for example, the production rule in G_{p-1} for generating *Expr* is recursive 100% of the time, expanding only to *Expr* "-" *Term*, without chance of hitting the base case. As a result, we take special measures to avoid such infinite productions during input generation (see Section 6.3.3).

```

+5 / -5 / 7 - +0 / 6 / 6 - 6 / 8 - 5 - 4
-4 / +7 / 5 - 4 / 7 / 4 - 6 / 0 - 5 - 0
+5 / ++4 / 4 - 8 / 8 - 4 / 8 / 7 - 8 - 9
-6 / 9 / 5 / 8 - +7 / -9 / 6 - 4 - 4 - 6
+8 / ++8 / 5 / 4 / 0 - 5 - 4 / 8 - 8 - 8
-9 / -5 / 9 / 4 - -9 / 0 / 5 - 8 / 4 - 6
++7 / 9 / 5 - +8 / +9 / 7 / 7 - 6 - 8 - 4
-+6 / -8 / 9 / 6 - 5 / 0 - 5 - 8 - 0 - 5

```

Figure 59: Inputs generated from G_{p-1} from Figure 58

If we use G_{p-1} as a production grammar—and avoiding infinite production—we obtain inputs as shown in Figure 59. These inputs now focus on operators like subtraction or division or unary operators not seen in our input samples. Likewise, the newly generated digits cover the complement of those digits previously seen. Yet, all inputs are syntactically valid according to the grammar.

In summary, with common inputs as produced by G_p , we can expect to have a good set of regression tests—or a set replicating the features of failure-inducing inputs when learning from failure-inducing samples. In contrast, uncommon inputs as produced by G_{p-1} would produce features rarely found in samples, and thus cover complementary functionality.

6.3 Approach

In order to explain our approach in detail, we start with introducing basic notions of probabilistic grammars.

6.3.1 Probabilistic Grammars

The probabilistic grammars that we employ in this chapter are based on the well-known context-free grammars (CFGs) [264].

Definition 1 (Context-free grammar). *A context-free grammar is a 4-tuple (V, T, P, S_0) , where V is the set of non-terminal symbols, T the terminals, $P : V \rightarrow (V \cup T)^*$ the set of productions, and $S_0 \in V$ the start symbol.*

In a *non-probabilistic grammar*, rules for a non-terminal symbol S provide n alternatives A_i for expansion:

$$S \rightarrow A_1 \mid A_2 \mid \dots \mid A_n \quad (2)$$

In a *probabilistic context-free grammar* (PCFG), each of the alternatives A_i in Equation (2) is augmented with a probability p_i , where $\sum_{i=1}^n p_i = 1$ holds:

$$S \rightarrow p_1 A_1 \mid p_2 A_2 \mid \dots \mid p_n A_n \quad (3)$$

If we are using these grammars for generation of a sentence of the language described by the grammar, each alternative A_i has a probability of p_i to be selected when expanding S .

By convention, if one or more p_i are not specified in a rule, we assume that their value is the complement probability, distributed equally over all alternatives with these unspecified probabilities. Consider the rule

Letter \rightarrow 40.0% "a" | "b" | "c"

Here, the probabilities for "b" and "c" are not specified; we assume that the complement of "a", namely 60%, is equally distributed over them, yielding effectively

Letter \rightarrow 40.0% "a" | 30.0% "b" | 30.0% "c"

Formally, to assign a probability to an unspecified p_i , we use

$$p_i = \frac{1 - \sum \{p_j | p_j \text{ is specified for } A_j\}}{\text{number of alternatives } A_k \text{ with unspecified } p_k} \quad (4)$$

Again, this causes the invariant $\sum_{i=1}^n p_i = 1$ to hold. If no p_i is specified for a rule with n alternatives, as in Equation (2), then Equation (4) makes each $p_i = 1/n$, as intended.

6.3.2 Learning Probabilities

Our aim is to turn a classical context-free grammar G into a probabilistic grammar G_p capturing the probabilities from a set of samples—that is, to determine the necessary p_i values as defined in Equation (3) from these samples. This is achieved by *counting* how frequently individual alternatives occur during parsing in each production context, and then to determine appropriate probabilities.

In language theory, the result of parsing a sample input I using G is a *derivation tree* [265], representing the structure of a sentence according to G . As an example, consider Figure 55, representing the input "1 + (2 * 3)" according to the example arithmetic expression grammar in Section 6.2. In this derivation tree, we can now *count* how frequently a particular alternative A_i was chosen in the grammar G during parsing. In Figure 55, the rule for *Expr* is invoked three times during parsing. This rule expands once (33.3%) into *Expr* "+" *Term* (at the root); and twice (66.7%) into *Term* in the subtrees. Likewise, the *Term* symbol expands once (25%) into *Term* "*" *Factor* and three times (75%) into *Factor*. Formally, given a set T of derivation trees from a grammar G applied on sample inputs, we determine the probabilities p_i for each alternative A_i of a symbol $S \rightarrow A_1 | \dots | A_n$ as

$$p_i = \frac{\text{Expansions of } S \rightarrow A_i \text{ in } T}{\text{Expansions of } S \text{ in } T} \quad (5)$$

If a symbol S does not occur in T , then Equation (5) makes $p_i = 0/0$ for all alternatives A_i ; in this case, we treat all p_i for S as *unspecified*, assigning them a value of $p_i = 1/n$ in line with Equation (4). In our example, Equation (5) yields the probabilistic grammar G_p in Figure 56.

6.3.3 Inverting Probabilities

We turn our attention now to the converse approach; namely producing inputs that *deviate* from the sample inputs that were used to learn the probabilities described above. This “uncommon input” approach promises to be useful if we accept that our samples are not able to cover all the

possible system behavior, and if we want to find bugs in behaviors that are either not exercised by our samples, or do so rarely.

The key idea is to *invert* the probability distributions as learned from the samples, such that the input generation focuses on the complement section of the language (w.r.t. the samples and those inputs generated by the probabilistic grammar). If some symbol occurs frequently in the parse trees corresponding to the samples, this approach should generate the symbol less frequently, and vice versa: if the symbol seldom occurs, then the approach should definitely generate it often.

For a moment, let us ignore probabilities and focus on *weights* instead. That is, the absolute (rather than relative) number of occurrences of a symbol in the parse tree of a sample. We start by determining the occurrences of a symbol A during a production S found in a derivation tree T :

$$w_{A,S} = \begin{array}{l} \text{Occurrence count of } A \text{ in the} \\ \text{expansions of symbol } S \text{ in } T \end{array} \quad (6)$$

To obtain *inverted* weights $w'_{A,S}$, a simple way is to make each $w'_{A,S}$ based on the reciprocal value of $w_{A,S}$, that is

$$w'_{A,S} = w_{A,S}^{-1} = \frac{1}{w_{A,S}} \quad (7)$$

If the set of samples is small enough, or focuses only on a section of the language of the grammar, it might be the case that some production or symbol never appears in the parsing trees. If this is the case, then the previous equations end up yielding $w_{A,S} = 0$. We can compute $w_{A,S}^{-1} = \infty$, assigning the elements not seen an infinite weight. Consequently, all symbols B that were indeed seen before (with $w_{B,S} > 0$) are assigned an infinitesimally small weight, leading to $w'_{B,S} = 0$. The remaining infinite weight is then distributed over all of the originally “unseen” elements with original weight $w_{A,S} = 0$. Recall the arithmetic expression grammar in Section 6.2; such a situation arises when we consider the rule for the symbol *Digit*: the inverted probabilities for the rule focus exclusively on the complement of the digits seen in the sample.

All that remains in order to obtain actual probabilities is to *normalize* the weights back into a probability measure, ensuring for each rule that its invariant $\sum_{i=1}^n p'_i = 1$ holds:

$$p'_i = \frac{w'_i}{\sum_{i=1}^n w'_i} \quad (8)$$

6.3.4 Producing Inputs from a Grammar

Given a probabilistic grammar G_p for some language (irrespective of whether it was obtained by learning from samples, by inverting, or simply written that way in the first place), our next step in the approach is to generate inputs following the specified productions. This generation process is actually very simple, since it reduces to produce instances by traversing the grammar, as if it were a Markov chain. However, this generation runs the serious risk of probabilistically choosing productions that lead to an excessively large parsing tree. Even worse, the risk of generating an *unbounded* tree is very real, as can be seen in the rule for the symbol *Int* in the arithmetic expression grammar in Section 6.2. The production rule for said symbol triggers, with probability 1.0, a recursion with no base case, and will never terminate.

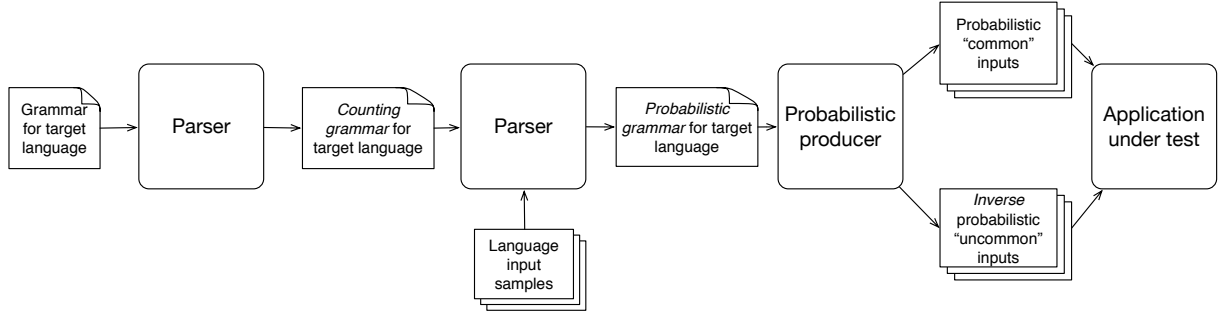


Figure 60: Workflow for the generation of “*common inputs*” and “*uncommon inputs*”

Our inspiration for constraining the growth of the tree during input generation comes from the PTC2 algorithm [266]. The main idea of this algorithm is to allow the expansion of not-yet-expanded productions, while ensuring that the number of productions does not exceed a certain threshold of performed expansions. This threshold would be set as parameter of the input generation process. Once this threshold is exceeded, the partially generated instance cannot be truncated, as that would result in an illegal input. Alternatively, we choose to allow further expansion of the necessary non-terminal symbols. However, from this point on, expansions are not chosen probabilistically. Rather, the choice is constrained to those expansions that generate the *shortest* possible expansion tree. This ensures both termination of the generation procedure, as well as trying to keep the input size close to the threshold parameter. This choice, however, does introduce a bias that may constitute a threat to the validity of our experiments that we discuss in Section 6.5.

6.3.5 Implementation

As a prerequisite for carrying out our approach, we only assume we have the context-free grammar of the language available for which we are interested in generating inputs, and a collection (no matter the size) of inputs that we will assume are *common* inputs. Armed with these elements, we perform the workflow detailed in Figure 60.

The first step of the approach is to obtain a *counting grammar* from the original grammar. This counting grammar is, from the parsing point of view, completely equivalent to the original grammar. However, it is augmented with *actions* during parsing which perform all necessary counting of symbol occurrences parallel to the parsing phase. Finally, it outputs the probabilistic grammar. Note that this first phase requires not only the grammar of the target language, but also the grammar of the *language in which the grammar itself is written*. That is, generating the probabilistic grammar not only requires parsing sample inputs, but also the grammar itself. In the particular case of our implementation, we make use of the well-known parser generator ANTLR [237].

Once the probabilistic grammar is obtained, we derive the probabilistically-inverted grammar as described in this section. Armed with both probabilistically annotated grammars, we can continue with the input generation procedure.

Table 17: Depth and size of derivation trees for “common inputs” (PROB) and “uncommon inputs” (INV)

Grammar	Mode	Depth of derivation tree				Nodes avg.
		min	max	avg.	median	
JSON	PROB	14	2867	96	63	3058
	INV	5	37	23	37	68
JavaScript	PROB	1	79	19	8	400
	INV	1	38	19	1	11,061
CSS	PROB	3	44	41	44	19,380
	INV	9	30	29	30	11,269

6.4 Experimental Evaluation

In this section we evaluate our approach by applying the technique in several case studies. In particular, we ask the following research questions:

- **RQ1 (“Common inputs”).** Can a learned grammar be used to generate inputs that resemble those that were employed during the grammar training?
- **RQ2 (“Uncommon inputs”).** Can a learned grammar be modified so it can generate inputs that, opposed to **RQ1**, are *in contrast* to those employed during the grammar training?
- **RQ3 (“Sensitivity to training set variance”).** Is our approach sensitive to variance in the initial samples?
- **RQ4 (“Sensitivity to size of training set”).** Is our approach sensitive to the size of the initial samples?
- **RQ5 (“Bugs found”).** What kind of crashes (exceptions) do we trigger in **RQ1** and **RQ2**?
- **RQ6 (“Failure-inducing inputs”).** Can a learned grammar be used to generate inputs that reproduce failure-inducing behavior?

To answer **RQ1** and **RQ2**, we need to compare inputs in order to decide whether these inputs are “similar” or “contrasting”. In the scope of this evaluation, we will use the *method coverage* and *call sequences* as measures of input similarity. We will define these measures later in this section, and we will discuss their usefulness. We address **RQ3** by comparing the method calls and call sequences induced for three randomly selected training sets, each containing five inputs. Likewise, we evaluate **RQ4** by comparing the method calls and call sequences induced for four randomly selected training sets, each containing N sample inputs, where $N \in \{1, 5, 10, 50\}$. We assess **RQ5** by categorizing, inspecting and reporting all exceptions triggered by our test suites in **RQ1** and **RQ2**. Finally, we address **RQ6** by investigating if the “(un)common inputs” strategy can reproduce (or avoid) a failure and explore the surroundings of the buggy behavior.

6.4.1 Evaluation Setup

Generated Inputs

Once a probabilistic grammar is learned from the training instances, we generate several inputs that are fed to each subject. Our evaluation involves the generation of three types of test suites:

- a) *Probabilistic* - choice between productions is governed by the distribution specified by the learned probabilities in the grammar.
- b) *Inverse* - choice is governed by the distribution obtained by the inversion process described in Section 6.3.3.
- c) *Random* - choice between productions is governed by a uniform distribution (see **RQ6**).

Expansion size control is carried out in order to avoid unbounded expansion as described in Section 6.3.4. Table 17 reports the details of the produced inputs, i.e. the depth and average number of nodes in the derivation trees for the “common inputs” (i.e., probabilistic/PROB) and “uncommon inputs” (i.e., inverse/INV).

Research Protocol

In our evaluation, we generate test suites and measure the frequency of method calls, the frequency of call sequences and the number of failures induced in our subject programs. For each input language, the experimental protocol proceeds as follows:

- a) We randomly selected five files from a pool of thousands of sample files crawled from GitHub code repositories, and through our approach produced a probabilistic grammar out of them⁵². Since one of the main use cases of our tool is to complete a test suite, we perform grammar training with few (i.e. five) initial sample tests.
- b) We feed the sampled input files into the subject program and record the triggered failures, the induced call sequences and the frequency of method calls using the HPROF [267] profiler for Java.
- c) Using the probabilistic grammar, we generate test suites, each one containing 100 input files. We generate a total of 1000 test suites, in order to control for variance in the input files. Overall, each experiment contains 100,000 input files (100 files x 1,000 runs). We perform this step for both probabilistic and inverse generations. Hence, the total number of inputs generated for each grammar is 200,000 (1,000 suites of 100 inputs each, a set of suites for each experiment).
- d) We test each subject program by feeding the input files into the subject program and recording the induced failures, the induced call sequences and the frequency of method calls using HPROF.

⁵²To evaluate **RQ6**, we learned a PCFG from at most five random failure-inducing inputs.

Table 18: Subject details

Input Format	Subject	Version	#Methods	LOC
JSON	Argo	5.4	523	8,265
	Genson	1.4	1,182	18,780
	Gson	2.8.5	793	25,172
	JSONJava	20180130	202	3,742
	Jackson	2.9.0	5,378	117,108
	JsonToJava	1880978	294	5,131
	MinimalJson	0.9.5	224	6,350
	Pojo	0.5.1	445	18,492
	json-simple	a8b94b7	63	2,432
	cliftonlabs	3.0.2	183	2,668
	fastjson	1.2.51	2,294	166,761
	json2flat	1.0.3	37	659
	json-flattener	0.6.0	138	1,522
JavaScript	Rhino	1.7.7	4873	100,234
	rhino-sandbox	0.0.10	49	529
CSS3	CSSValidator	1.0.4	7774	120,838
	flute	1.3	368	8,250
	jstyleparser	3.2	2,589	26,287
	cssparser	0.9.27	2,014	18,465
	closure-style	0.9.27	3,029	35,401

All experiments were conducted on a server with 64 cores and 126 GB of RAM; more specifically an Intel Xeon CPU E5-2683 v4 @ 2.10GHz with 64 virtual cores (Intel Hyperthreading), running Debian 9.5 Linux.

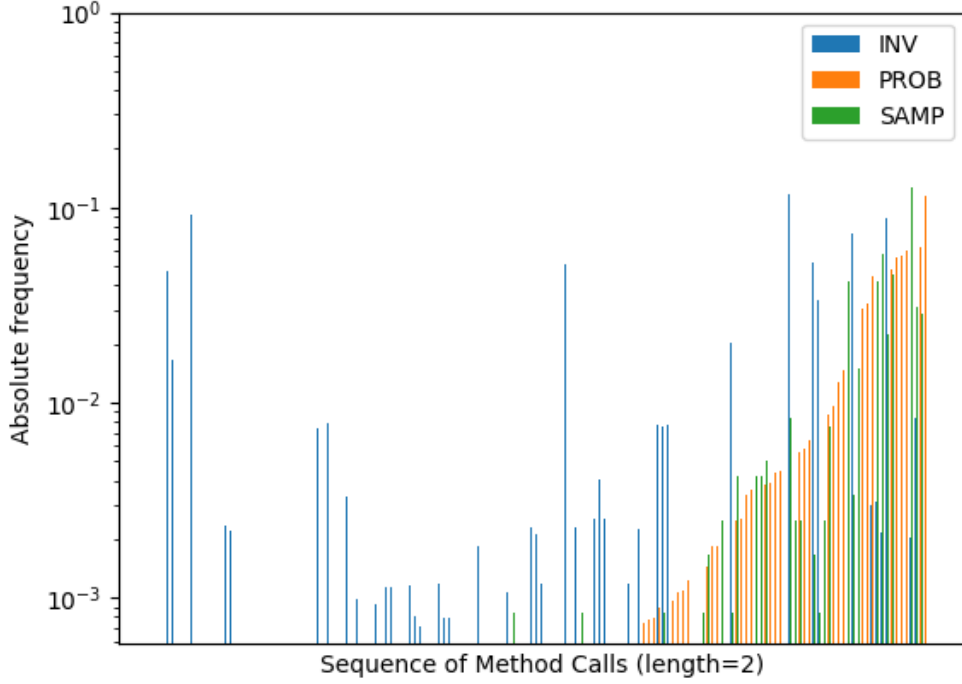
Subject Programs

We evaluated our approach by generating inputs and feeding them to a variety of Java applications. All these applications are open source programs using three different input formats, namely JSON, JavaScript and CSS3. Table 18 summarizes the subjects to be analyzed, their input format and the number of methods in each implementation.

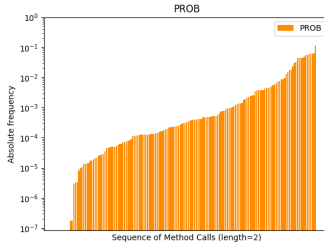
The initial, unquantified grammars for the input subjects were adapted from those in the repository of the well-known parser generator ANTLR [237]. We handle grammar ambiguity that may affect learning probabilities by ensuring every input has only one parse tree. Specifically, we adapt the input grammars by (re-)writing lexer modes for the grammars, shortening lexer tokens and re-writing parser rules. Training samples were obtained by scraping GitHub repositories for the required format files. The probabilistic grammars developed from the original ones, as well as the obtained training samples can be found in our replication package.

Measuring (Dis)similarity

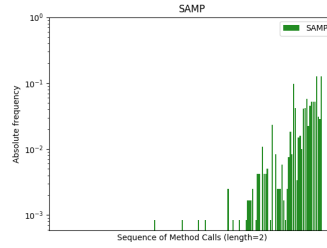
Questions **RQ1** and **RQ2** refer to a notion of similarity between inputs. Although white-box approaches exist that aim to measure test-case similarity and dissimilarity [268, 269], applying them to complex grammar-based inputs is not straightforward. However, in this work, since we are dealing with evaluating the behavior of a certain piece of software, it makes sense to aim for



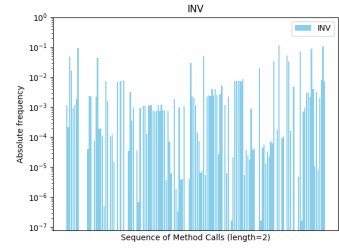
(a) PROB vs. SAMP vs. INV



(b) PROB



(c) SAMP



(d) INV

Figure 61: Frequency analysis of call sequences for json-flattener (length=2)

a notion of *semantic* similarity. In this sense, two inputs are semantically similar if they incite similar behaviors in the software that processes them. In order to achieve this, we define two measures of input similarity based on structural and non-structural program coverage metrics. The *non-structural* measure of input similarity is the *frequency of method calls* induced in the programs. The *structural* measure is the *frequency of call sequences* induced in the program, a similar measure was used in [270]. Thus, we will say two inputs are similar if they induce similar (distribution of) method call frequencies for the same program. The *frequency of call sequences* refers to the number of times a specific method call sequence is triggered by an input, for a program. For this measure, we say two inputs are similar if they trigger a similar distribution in the frequency with which the method sequences are called, for the same program. These notions allow for a great variance drift if we were to compare only two inputs. Therefore, we perform these comparisons on test suites as a whole to dampen the effect of this variance. Using these measures, we aim at answering **RQ1** and **RQ2**. **RQ1** will be answered satisfactorily if the (distribution of) call frequencies and sequences induced by the “common inputs” strategy is

Table 19: Call Sequence analysis for “common inputs” (PROB) and “uncommon inputs” (INV) for all subject programs

Length	Call Sequences covered by Sample			Call sequences covered by	
	#	also by PROB	also by INV	PROB	INV
2	1210	1157 (96%)	937 (74%)	6348	5196
3	1152	1099 (95%)	782 (62%)	7946	5930
4	849	803 (90%)	479 (47%)	9236	5825
Total	3211	3059 (94%)	2198 (61%)	23 530	16 951

similar to the call frequency and sequences obtained when running the software on the *training samples*. Likewise, **RQ2** will be answered positively if the (distribution of) call frequencies and sequences for suites generated with the “uncommon inputs” strategy are markedly *dissimilar*.

Visual test

For each test suite, we compare the frequency distribution of the call sequences and method calls triggered in a program, using grouped and single bar charts. These comparisons are in line with the visual tests described in [271].

For instance, Figure 61 shows the frequency analysis of the call sequences induced in `json-flattener` by our test suites. The grouped bar chart compares the frequency distribution of call sequences for all three test suites, (i.e. (a) PROB vs. SAMP vs. INV) and the single bar chart shows the frequency distribution of call sequences for each test suite (i.e., (b.) PROB, (c) SAMP and (d) INV). Frequency analysis (in (a.)) shows that the (distribution of) call sequences of PROB and SAMP align (see rightmost part of bar chart), and INV often induces a different distribution of call sequences from the initial samples (see leftmost part of bar chart). The single bar chart for a test suite shows the frequency distribution of the call sequences triggered by the test suite. For instance, Figure 61 (b) and (c) show the call sequence distribution triggered by the “common inputs” and initial samples respectively. The comparison of both charts shows that all call sequences covered by the samples, were also frequently covered by the “common inputs”.

Likewise, Figure 63 to Figure 65 show the call frequency analysis of the test suites using a grouped bar chart for comparison (i.e. (a) PROB vs. SAMP vs. INV) and a single bar chart to show the call frequency distribution of each test suite (i.e., (b.) PROB, (c) SAMP and (d) INV). The grouped bar chart shows the call frequency for each test suite grouped together by method, with bars for each test suite appearing side by side per method. For instance, analysing Figure 63 (a) shows that the call frequencies of PROB and SAMP align (see rightmost part of bar chart), and INV often induces a different call frequency for most methods (see leftmost part of bar chart). Moreover, the single bar chart for a test suite shows the call frequency distribution of the test suite. For instance, Figure 63 (b) and (c) show the call frequency distribution of the “common inputs” and initial samples respectively, their comparison shows that all methods covered by the samples, were also frequently covered by the “common inputs”.

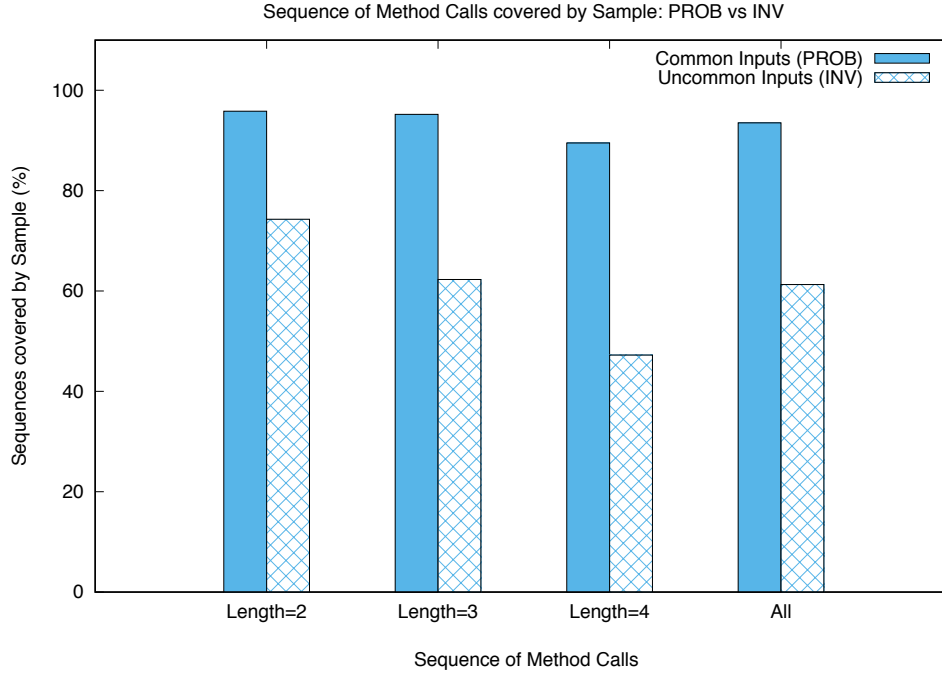


Figure 62: Call sequences covered by Sample for “common inputs” (PROB) and “uncommon inputs” (INV)

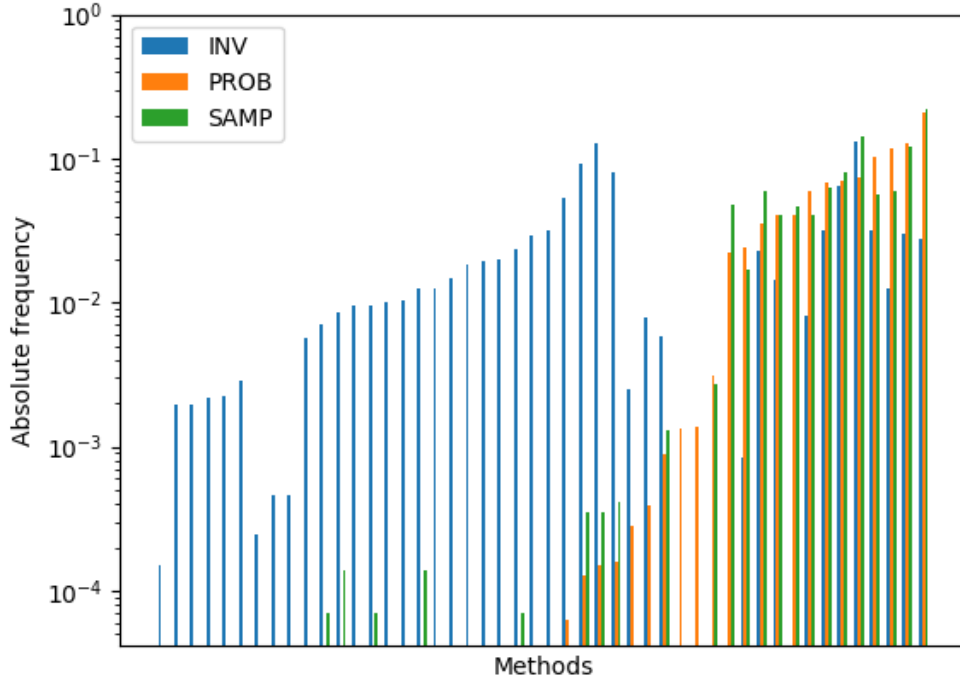
Collecting failure-inducing inputs

For each input file in our Github corpus, we fed it to every subject program of the input language and observe if the subject program crashes, i.e. the output status code is non-zero after execution. Then, we collect such inputs as failure-inducing inputs for the subject program and parse the standard output for the raised exception. In total, we fed 10,853 files to the subject programs, 1,000 each for CSS and JavaScript, and 8853 for JSON. Exceptions were triggered for two input languages, namely JavaScript and JSON, no exception was triggered for CSS. In total, we collected 15 exceptions in seven subject programs (*see Table 26*).

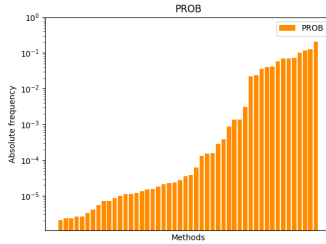
6.4.2 Experimental Results

In Figure 63 to Figure 65, we show a representative selection of our results⁵³. For each subject, we constructed a chart that represents the absolute call frequency of each method in the subject. The horizontal axis (which is otherwise unlabelled) represents the set of methods in the subject, ordered by the *frequency of calls* in the experiment on *probabilistic inputs*.

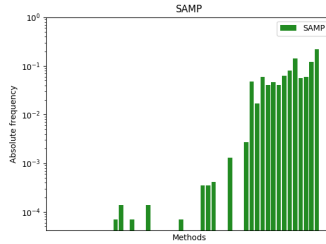
⁵³The full range of charts is omitted for space reasons. However, all charts, as well as the raw data, are available as part of the artifact package. Moreover, the charts shown here have been selected so that they are representative of the whole set; that is, the omitted charts do not deviate significantly.



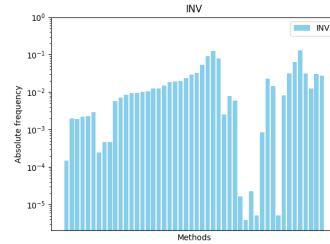
(a) PROB vs. SAMP vs. INV



(b) PROB



(c) SAMP



(d) INV

Figure 63: Call frequency analysis for json-simple-cliftonlabs

RQ1 (“Common inputs”): Can a learned grammar be used to generate inputs that resemble those that were employed during the grammar training?

To answer **RQ1**, we compare the methods covered by the sample inputs and the “common inputs” strategy (Table 20 and Figure 63 to Figure 65). We also examine the call sequences covered by the sample inputs and the “common inputs”, for consecutive call sequences of length two, three and four (Table 19 and Figure 62). In particular, we investigate if the “common inputs” strategy covered at least the same methods or the same call sequences as the initial sample inputs.

Do the “common inputs” trigger similar non-structural program behavior (i.e., method calls) as the initial samples? For all subjects, the “common inputs” strategy covered almost all (96%) of the methods covered by the sample (see Table 20). This result shows that the “common inputs” strategy learned the input properties in the samples and reproduced the same (non-structural) program behavior as the initial samples. Besides, this strategy also covered other methods that were not covered by the samples.

Table 20: Method coverage for “common inputs” (PROB) and “uncommon inputs” (INV)

Subject	#	Methods covered by sample also by		Methods covered by		Kolmogorov-Smirnov (KS) test of SAMP vs. PROB	
		PROB	INV	PROB	INV	D-stat (p-value)	SAMP vs. INV D-stat (p-value)
Argo	52	52 (100%)	32 (62%)	256	165	0.28 (1.11E-9)	0.50 (1.84E-30)
Genson	12	12 (100%)	10 (83%)	218	188	0.25 (3.46E-7)	0.73 (5.88E-64)
Gson	24	24 (100%)	14 (58%)	287	239	0.52 (1.09E-40)	0.25 (1.94E-9)
JSONJava	29	29 (100%)	23 (79%)	51	42	0.08 (0.99)	0.63 (3.45E-11)
Jackson	2	2 (100%)	1 (50%)	957	732	N/A	N/A
JsonToJava	29	29 (100%)	9 (31%)	82	33	0.25 (8.48E-3)	0.24 (1.38E-2)
MinimalJson	24	24 (100%)	18 (75%)	110	100	0.34 (2.36E-6)	0.83 (5.39E-42)
Pojo	23	23 (100%)	7 (30%)	159	93	0.19 (1.81E-3)	0.29 (1.04E-7)
json-simple	11	11 (100%)	10 (91%)	26	24	0.35 (0.09)	0.46 (7.13E-3)
cliftonlabs	23	23 (100%)	23 (100%)	48	48	0.21 (0.25)	0.54 (8.29E-7)
fastjson	70	70 (100%)	62 (89%)	245	231	0.37 (3.07E-15)	0.41 (8.84E-19)
json2flat	6	6 (100%)	5 (83%)	17	14	0.35 (0.24)	0.65 (1.15E-3)
json-flattener	36	36 (100%)	32 (89%)	83	81	0.15 (0.29)	0.15 (0.29)
Rhino	23	6 (26%)	23 (100%)	107	201	0.34 (3.18E-12)	0.45 (6.48E-21)
rhino-sandbox	3	3 (100%)	3 (100%)	17	17	0.47 (0.04)	0.53 (0.02)
CSSValidator	10	10 (100%)	10 (100%)	97	124	0.42 (1.20E-10)	0.38 (7.95E-9)
flute	58	57 (98%)	51 (88%)	148	131	0.29 (3.35E-6)	0.50 (7.34E-18)
jstyleparser	75	74 (99%)	59 (79%)	183	169	0.34 (1.53E-13)	0.38 (1.83E-17)
cssparser	71	71 (100%)	66 (93%)	177	152	0.36 (2.91E-12)	0.62 (4.21E-37)
closure-style	104	95 (91%)	103 (99%)	229	238	0.16 (2.03E-6)	0.09 (3.34E-2)
Total	685	657 (96%)	561 (82%)	3,497	3,022		

*The “common inputs” strategy triggered almost all methods (96%)
called by the initial sample inputs.*

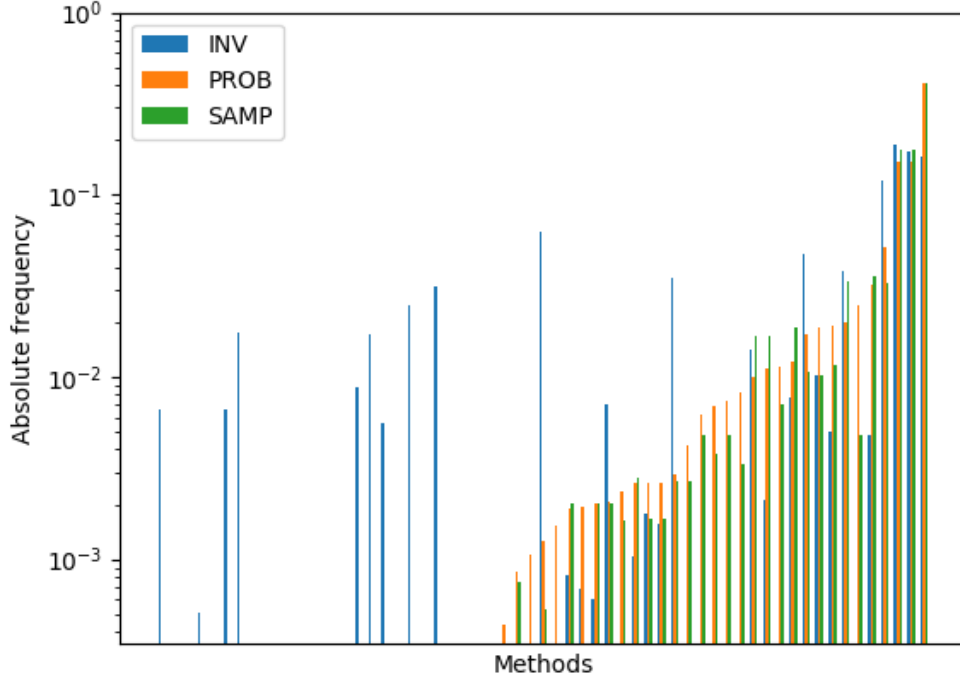
Do the “common inputs” also trigger similar structural program behavior (i.e., sequences of method calls)? In our evaluation, the “common inputs” strategy covered most of the call sequences that were covered by the initial samples. For instance, Figure 61 shows that the call sequences covered by the samples were also frequently covered by the “common inputs”, for `json-flattener`. Overall, the “common inputs” strategy covered 94% of the method call sequences induced by the sample (see Table 19 and Figure 62). For all call sequences, the “common inputs” strategy also covered 90% to 96% of the method call sequences covered by the samples. This result shows that the ‘common inputs’ strategy triggers the same structural program behavior as the initial samples.

*The “common inputs” strategy triggered most call sequences (94%)
covered by the initial sample inputs.*

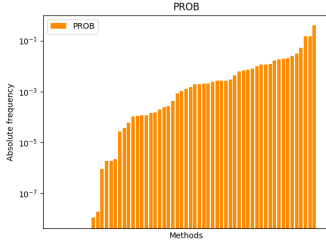
Additionally, we compare the statistical distributions resulting from our strategies. We need to be able to see a pattern in frequency calls such that the frequency curves for the *sample* runs and the *probabilistic* runs match as described in the visual test (see Section 6.4.1). Figure 63 to Figure 65 show that this match does hold for all subjects.

*For all subjects, the method call frequency curves for the sample runs
and the probabilistic runs match.*

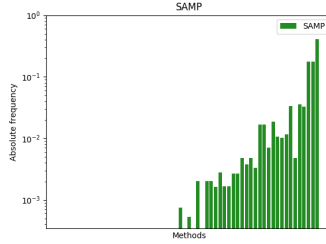
We also perform a statistical analysis on the distributions to increase the confidence in our conclusion. We performed a distribution fitness test (KS - Kolmogorov-Smirnov) on the sample vs. the probabilistic call distribution; and on the sample vs. the inverse probabilistic distribution. It must be noted that the KS test aims at determining whether the distributions are *exactly* the



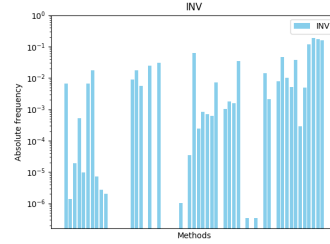
(a) PROB vs. SAMP vs. INV



(b) PROB



(c) SAMP

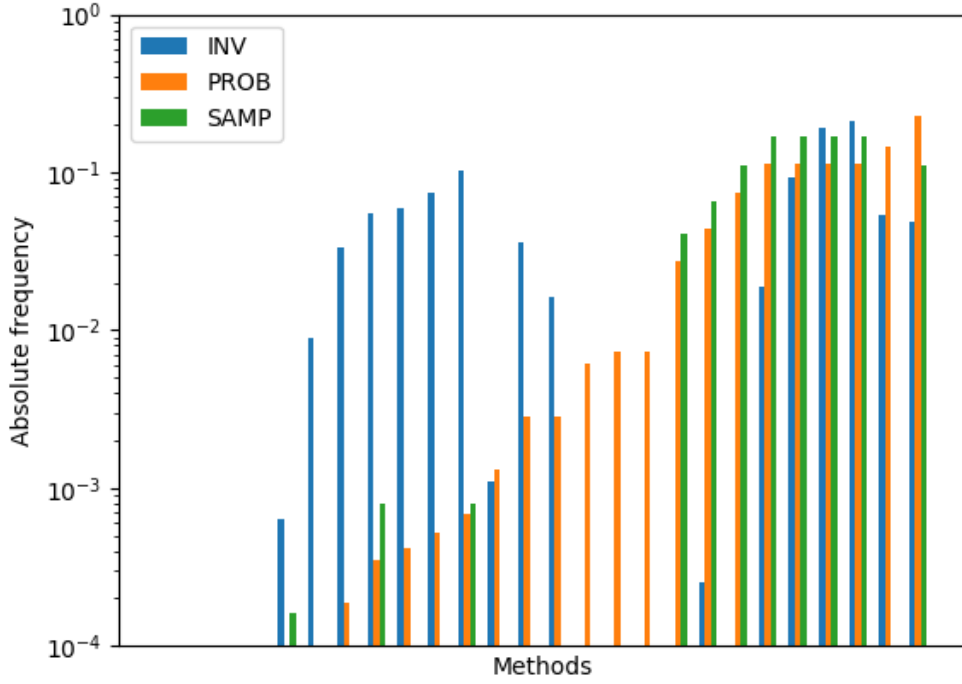


(d) INV

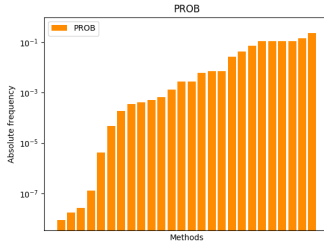
Figure 64: Call frequency analysis for JSONJava

same, whereas we want to ascertain if they are *similar* or *dissimilar*. KS tests are *very sensitive* to small variations in data, which makes it, in principle, inadequate for this objective. In this work, we employ the approach used by Fan [272]—we first estimate the kernel density functions of the data distributions, which smoothen the estimated distribution. Then, we bootstrap and resample new data on the kernel density estimates, and perform the KS test on the bootstrapped data.

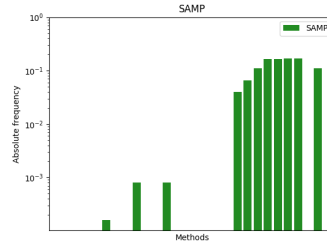
The KS test confirms the results from the visual inspection, the distribution of the method call frequency of “common inputs” matches the distribution in the sample (*see Table 20*), for some subjects. However, there are also subjects, where the hypothesis is rejected ($p < 0.05$) that method call frequency distributions (sample and “common inputs”) come from the same distribution, which is indicated by the blue entries. In the case of the Jackson subject, frequencies for the sample calls are all close to zero, which makes the data inadequate for the KS test.



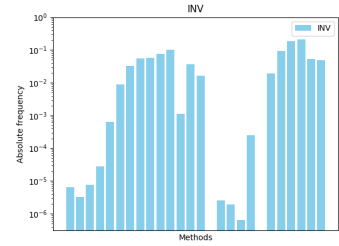
(a) PROB vs. SAMP vs. INV



(b) PROB



(c) SAMP



(d) INV

Figure 65: Call frequency analysis for json-simple

RQ2 (“Uncommon inputs”): Can a learned grammar be modified such it can generate inputs that, opposed to RQ1, are *in contrast* to those employed during the grammar training?

For all subjects, the “uncommon inputs” produced by inverting probabilities covered markedly fewer (82%) of the methods covered by the sample (see Table 20). This result shows that the “uncommon inputs” strategy learned the input properties in the samples and produced inputs that avoid several methods covered by the samples.

The “uncommon inputs” strategy triggered markedly fewer methods (82%) called by the initial sample inputs.

Do the “uncommon inputs” trigger fewer of the call sequences covered by the initial samples? Table 19 shows that the “uncommon inputs” strategy triggered significantly fewer (61%) of the call sequences covered by the samples. The number of call sequences induced by the uncommon

Table 21: Sensitivity to training set variance using three different sets ($S \in \{1, 2, 3\}$) of initial samples containing five inputs each

S	Methods covered by Sample also by				Methods covered by		Call Seq. covered by Sample also by			Call Seq. covered by				
	#	PROB		INV	PROB	INV	#	PROB	INV	PROB	INV			
1	685	657	(96%)	561	(82%)	3,497	3,022	3,211	3,059	(94%)	2,198	(61%)	23,530	16,951
2	2,963	2,924	(97%)	2,764	(85%)	8,623	8,246	6,044	5,643	(93%)	4,110	(68%)	22,531	19,896
3	2,656	2,639	(100%)	2,516	(87%)	8,655	8,165	5,005	4,915	(98%)	3,306	(66%)	20,792	19,892

Table 22: Sensitivity to the size of the training set using initial sample size $N \in \{1, 5, 10, 50\}$

N	Methods covered by Sample also by			Methods covered by		Call Seq. covered by Sample also by			Call Seq. covered by	
	#	PROB	INV	PROB	INV	#	PROB	INV	PROB	INV
1	1,496	1,490 (100%)	1,352 (79%)	8,715	7,954	1,955	1,942 (99%)	1,135 (58%)	21,279	15,341
5	685	657 (96%)	561 (82%)	3,497	3,022	3,211	3,059 (94%)	2,198 (61%)	23,530	16,951
10	3,546	3,517 (100%)	3,339 (89%)	9,388	11,497	6,297	6,105 (97%)	4,474 (71%)	26,575	21,214
50	5,347	5,313 (100%)	4,961 (89%)	8,950	8,217	9,389	9,076 (97%)	7,421 (79%)	23,512	18,391

inputs decreases significantly as the length of the call sequence increases (*see Figure 62*). For instance, comparing frequency charts of call sequences in Figure 61 ((a), (c) and (d)) also show that “uncommon inputs” frequently avoided inducing the call sequences triggered by the initial samples. Notably, for sequences of four consecutive method calls, the “uncommon inputs” strategy covered only 47% of the sequences covered by the initial samples (*see Table 19*). Overall, the “uncommon inputs” avoided inducing the call sequences that were triggered by the initial samples.

The “uncommon inputs” strategy induced significantly fewer call sequences (61%) covered by the initial samples.

Do the “uncommon inputs” only cover *fewer*, or also *different* methods? We perform a visual test to examine if we see a *markedly different* call frequency between the samples and the inputs generated by the “uncommon inputs” strategy. In almost all charts this is the case (*see Figure 63 to Figure 65*). The only exception is the CSSValidator subject.

For all subjects (except CSSValidator), the method call frequency curves for the sample runs and “uncommon inputs” runs are markedly different.

Besides, we examine if the frequency of distribution of method calls for the samples and the “uncommon inputs” are *significantly dissimilar*. In particular, the KS tests shows that for all subjects (except json-flattener) the distributions of method calls in the sample and the “uncommon inputs” are significantly different ($p < 0.05$, *see sample vs. INV in Table 20*).

RQ3 (“Sensitivity to training set variance”): Is our approach sensitive to variance in the initial samples?

We examine the sensitivity of our approach to the variance in the training set. We randomly selected three distinct training sets, each containing five inputs. Then, for each set, we compare the methods and call sequences covered by the samples to those induced by the generated inputs (Table 21).

Our evaluation showed that *our approach is not sensitive to training set variance*. In particular, for all training sets, the “common inputs” strategy covered most of the methods and call sequences

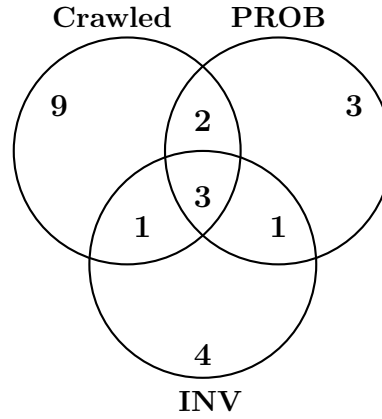


Figure 66: Number of exceptions triggered by the test suites

induced by the initial sample inputs. Table 21 shows that the “common inputs” (PROB) consistently covered almost all call sequences (93 to 98%) covered by the initial samples, while “uncommon inputs” (INV) covered significantly fewer call sequences (61 to 68%). Likewise, the “common inputs” consistently covered almost all methods (96 to 100%) covered by the initial samples, while “uncommon inputs” covered fewer methods (82 to 87%) (*cf. Table 21*).

Both strategies, the “common inputs” and the “uncommon inputs”, are insensitive to training set variance.

RQ4 (“Sensitivity to the size of training set”): Is our approach sensitive to the size of the initial samples?

We evaluate the sensitivity of our approach to the size of the training set. For each input format, we randomly selected four distinct training sets containing N sample inputs, where $N \in \{1, 5, 10, 50\}$. Then, for each set, we compare the methods and call sequences induced by the samples to those induced by the generated inputs (Table 22).

Regardless of the size of the training set, the “common inputs” strategy consistently covered most of the methods and call sequences covered by the initial samples. Specifically, for all sizes, the “common inputs” covered almost all (94 to 99%) of the call sequences covered by the initial samples, while “uncommon inputs” covered significantly fewer call sequences (58 to 79%). In the same vein, the “common inputs” consistently covered almost all methods (96 to 100%) covered by the initial samples, while “uncommon inputs” covered fewer methods (79 to 89%) (*cf. Table 22*). These results demonstrates that the effectiveness of our approach is independent of the size of the training set.

The effectiveness of our approach is independent of the size of the training set used for grammar training.

RQ5 (“Bugs found”): What kind of crashes (exceptions) do we trigger?

To address **RQ5**, we examine all of the exceptions triggered by our test suites. We inspect the exceptions triggered during our evaluation of the “common inputs” strategy (in **RQ1**) and

Table 23: Exception details

Input	#Exceptions	#Subjects	Average subject crash rate	
			(All)	(Crashed)
SAMP	3	1	0.05263	1
PROB	9	4	0.05999	0.28493
INV	9	7	0.03139	0.08521

the “uncommon inputs” strategy (in **RQ2**). To evaluate if our approach is capable of finding real-world bugs, we compare the exceptions triggered in both **RQ1** and **RQ2** to the exceptions triggered by the input files crawled from *Github* (using the setup described in Section 6.4.1).

Both of our strategies triggered 40% of the exceptions triggered by the crawled files, i.e. six (out of 15) exceptions causing thousands of crashes in four subjects (*cf. Table 23 and Table 24*). Half (three) of these exceptions had no samples of failure-inducing inputs in their grammar training. This indicates that, even without failure-inducing input samples during grammar training, our approach is able to trigger *common buggy behaviors* in the wild, i.e. bugs triggered by the crawled input samples. Exceptions were triggered for JSON and JavaScript input formats, however, no exception was triggered for CSS.

Probabilistic grammar-based testing induced two fifths of all exceptions triggered by the crawled files.

Our strategies were able to trigger eight other exceptions that could not be found by the crawled files (*cf. Figure 66*). This result shows the utility of our approach in finding *rare buggy behaviors*, i.e. uncommon bugs in the crawled input samples. Besides, all of these exceptions were triggered despite a lack of “failure-inducing input” samples in the grammar training. In particular, both strategies triggered nine exceptions each, three and four of which were triggered only by the “common inputs” and only by the “uncommon inputs”, respectively.

Probabilistic grammar-based testing induced eight new exceptions that were not triggered by the crawled files.

The “common inputs” strategy triggered all of the exceptions triggered by the original sample inputs used in grammar training. Three exceptions were triggered by the sample inputs and all three exceptions were triggered by the “common inputs” strategy, while “uncommon inputs” triggered only two of these exceptions (*cf. Table 23 and Table 24*). Again, this result confirms that our grammar training approach can learn the input properties that characterize specific program behaviors.

The “common inputs” induced all of the exceptions triggered by the original sample inputs.

Overall, 14 exceptions in seven subject programs were found in our experiments (*see Table 23 and Table 24*). On inspection, six of these exceptions affecting five subject programs have been reported to developers as severe bugs. These exceptions have been extensively discussed in the bug repository of each subject program. This result reveals that our approach can generate inputs that reveal real-world buggy behaviors. Additionally, from the evaluation of crawled files, 15 exceptions in five subjects were found. In particular, one exception (Rhino issue #385, which is also reproducible with our approach) has been confirmed and fixed by the developers.

Table 24: Exceptions induced by “Common Inputs” (PROB), and “Uncommon Inputs” (INV)

Input Format	Subject	Exception	#Failure-inducing samples	Occurrence rate in		
				PROB	INV	Crawled Files
CSS	No exceptions triggered					
JSON	Gson	java.lang.NullPointer	0	0	0.0001	0
	Pojo	java.lang.StringIndexOutOfBounds	0	0.0259	0	0.0024
	Pojo	java.lang.NumberFormat	0	0	0.0001	0
	Argo	argo.saj.InvalidSyntax	0	0	0.0023	0.0225
	JSONToJava	org.json.JSON	0	0.0200	0.0200	0.0223
	json2flat	com.fasterxml.jackson.core.JsonParse	0	0	0.0013	0
	json-flattener	com.eclipsesource.json.Parse	0	0	0.0013	0
	json-flattener	java.lang.UnsupportedOperation	0	0.2981	0	0
	json-flattener	java.lang.IllegalArgumentException	0	0.2554	0	0
	json-flattener	java.lang.NullPointer	0	0.0398	0	0
	json-flattener	java.lang.NumberFormat	0	0.0028	0.0745	0
Java-Script	rhino-sandbox	org.mozilla.javascript.Evaluator	3	0.4905	0.4469	0.5290
	rhino-sandbox	java.lang.IllegalState	1	0.0016	0	0.0010
	rhino-sandbox	org.mozilla.javascript.EcmaError	1	0.0058	0.0500	0.3740

RQ6 (“Failure-inducing inputs”): Can a learned grammar be used to generate inputs that reproduce failure-inducing behavior?

Let us now investigate if our approach can learn a PCFG from failure-inducing samples and reproduce the failure.

For each exception triggered by the crawled files (in Section 6.4.1), we learned a PCFG from at most five failure-inducing inputs that trigger the exception. Then, we run our PROB approach on the PCFG, using the protocol setting in Section 6.4.1. The goal is to demonstrate that the PCFG learns the input properties of the “failure-inducing inputs”, i.e. inputs generated via PROB should trigger *the same exception* as the failure-inducing samples. This is useful for exploring the surroundings of a bug.

In addition, for each exception, we run the inverse of “failure-inducing inputs” (i.e., INV), in order to evaluate if the “uncommon inputs” avoid reproducing the failures. In contrast, for each exception, we run the random generator (RAND) on the CFG (according to Section 6.4.1), in order to evaluate the probability of randomly triggering (these) exceptions without a (learned) PCFG. In the random configuration (RAND), production choices have equal probability of generation.

In Table 25, we have summarized the number of reproduced exceptions. We see that probabilistic generation (PROB) reproduced all (15) failure-inducing inputs collected in our corpus. This shows that the grammar training approach effectively captured the distribution of input properties in the failure-inducing inputs. Moreover, it reproduced the program behavior using the learned PCFG.

*Learning probabilities from failure-inducing inputs strategy reproduces
100% of the exceptions in our corpus.*

For the inverse of “failure-inducing inputs”, our evaluation showed that the “uncommon inputs” strategy could avoid reproducing the learned failure-inducing behavior for most (10 out of 15) of the exceptions (*cf.* Table 25 and Table 26).

*The “uncommon inputs” strategy could avoid reproducing the learned program behavior
for two-thirds of the exceptions.*

Table 25: Reproduced exceptions by sample inputs (SAMP), “failure-inducing inputs” (PROB), inverse of “failure-inducing inputs” (INV) and random grammar-based generation (RAND)

	#Exceptions		Average # Failure-inducing inputs
	Reproduced	Other	
SAMP	15	0	87
PROB	15	21	18,429
INV	5	6	18,080
RAND	0	0	0

However, this strategy reproduced a third (five out of 15) of the exceptions in our corpus (*cf.* Table 25 and Table 26). On inspection, we found that “uncommon inputs” reproduced these five exceptions by generating new counter-examples, i.e., new inputs that are different from the initial samples but trigger the same exception. This is because the initial sample of failure-inducing inputs was not general enough to fully characterize all input properties triggering the crash. This result demonstrates that the inverse of “failure-inducing inputs” can explore the boundaries of the learned behavior in the PCFG, hence, it is useful for generating counter-examples.

*The “uncommon inputs” strategy generated new counter-examples
for one-third of the exceptions in our corpus.*

In contrast, the random test suite (RAND) could not trigger *any* of the exceptions in our corpus, as shown in Table 25. This is expected, since a random traversal of the input grammar would need to explore numerous path combinations to find the specific paths that trigger an exception. This result demonstrates the effectiveness of the grammar training and the importance of the PCFG in capturing input properties.

Random input generation could not reproduce any of the exceptions in our corpus.

Furthermore, we examined the proportion of the generated inputs that trigger an exception. In total, for each test configuration and each exception we generated 100,000 inputs. We investigate the proportion of these inputs that trigger the exception.

Our results for this analysis are summarized in Table 26. We see that about 18% of the inputs generated by the “failure-inducing inputs” strategy (PROB) trigger the learned exception, on average. This rate is three times as high as the exception occurrence rate in our corpus (SAMP; 6%).

*About one in five inputs generated by the “failure-inducing inputs” strategy
reproduced the failure-inducing exception.*

Finally, the “failure-inducing inputs” strategy also produced *new exceptions* not produced by the original sample of failure-inducing inputs. As shown in the “Other” column in Table 25, “failure-inducing inputs” triggered at least one new exception for each exception in our corpus. This result suggests that the PCFG is also useful for exploring the boundaries of the learned behavior, in order to trigger other program behaviors different from the learned program behavior. This is possible because “failure-inducing inputs” not only reproduces the exact features found in the samples, but also their variations and combinations.

Table 26: Reproducing exceptions by “failure-inducing inputs” (PROB), inverse of “failure-inducing inputs” (INV), and random grammar-based test generation (RAND), showing (A) the *number of failure-inducing inputs* in crawled files, (B) the *exception occurrence rate in crawled files*, as well as the *exception reproduction rate in* (C) **PROB**, (D) **RAND** and (E) **INV**

Input Format (#files)	Subject	Exception	A	B	Reproduction rate in		
					C	D	E
JSON (8853)	Gson	java.lang.ClassCast	6	0.0007	0.0090	0	0
	Gson	java.lang.IllegalState	22	0.0025	1	0	1
	JSONToJava	java.lang.ArrayIndexOutOfBoundsException	38	0.0043	0.0025	0	0
	JSONToJava	java.lang.IllegalArgumentException	1	0.0001	0.0003	0	0
	JSONToJava	org.json.JSON_1	167	0.0189	0.1811	0	0.1811
	JSONToJava	org.json.JSON_2	30	0.0034	1	0	1
	Pojo	java.lang.IllegalArgumentException	88	0.0099	0.0002	0	0
	Pojo	java.lang.StringIndexOutOfBoundsException	21	0.0024	0.0471	0	0
JavaScript (1000)	Rhino	java.util.concurrent.Timeout	11	0.0110	0.0048	0	0
	Rhino	java.lang.IllegalState	2	0.0030	0.0001	0	0
	rhino-sandbox	delight.rhinosandbox.ScriptDuration	11	0.0110	0.0073	0	0
	rhino-sandbox	org.mozilla.javascript.Evaluator	529	0.5290	0.4560	0	0.4982
	rhino-sandbox	org.mozilla.javascript.EcmaError_1	372	0.3720	0.0056	0	0.0326
	rhino-sandbox	org.mozilla.javascript.EcmaError_2	2	0.0020	0.0002	0	0
	rhino-sandbox	org.mozilla.javascript.JavaScript	1	0.0010	0.0502	0	0
AVERAGE				0.0646	0.1842	0	0.1808

The “failure-inducing inputs” strategy discovered new exceptions not triggered by the samples or random generation.

6.5 Threats to Validity

Internal Validity

The main threat to internal validity is the correctness of our implementation. Namely, whether our implementation does indeed learn a probabilistic grammar corresponding to the distribution of the real world samples used as training set. Unfortunately, this problem is not a simple one to resolve. The probabilistic grammar can be seen as a Markov chain, and the aforementioned problem is equivalent to verifying that its equilibrium distribution corresponds to the posterior distribution of the real world samples. The problem is two-fold: first, the number of samples necessary in order to ascertain the posterior distribution is inordinate. Second, even if we had a chance to process such a number of inputs, or if the posterior distribution were otherwise known, it might well be the case that the probabilistic grammar actually has no equilibrium distribution. However, our tests on smaller and simpler grammars suggest that this is not an issue.

A second internal validity threat is present in the technique we use for controlling the size of the generated samples. As described before, a sample’s size is defined in terms of the number of expansions in its parsing tree. In order to control the size, we keep track of the number of expansions generated. Once this number crosses a certain threshold (if it actually crosses it at all), all open derivations are closed via their shortest path. This does introduce a bias in the generation that does not exactly correspond to the distribution described by the probabilistic grammar. The effects of such a bias are difficult to determine, and merit further and deeper study. However, not performing this termination procedure would render useless any approach based on probabilistic grammars.

External Validity

Threats to external validity relate to the generalizability of the experimental results. In our case, this is specifically related to the subjects used in the experiments. We acknowledge that we have only experimented with a limited number of input grammars. However, we have selected the subjects with the intention to test our approach on practically relevant input grammars with different complexities, from small to medium size grammars like JSON; and rather complex grammars like JavaScript and CSS. As a result, we are confident that our approach will also work on inputs that can be characterized by context-free grammars with a wide range of complexity. However, we do have evidence that the approach does not seem to be generalizable to combinations of grammars and samples such that they induce the learning of an almost-uniform probabilistic grammar.

Construct Validity

The main threat to construct validity is the metric we use to evaluate the similarity between test suites, namely method call frequency. While the uses of coverage metrics as adequacy criteria is extensively discussed by the community [66, 273, 274], their binary nature (that is, we can either report *covered* or *not covered*) makes them too shallow to differentiate for behavior. The variance intrinsic to the probabilistic generation makes it very likely that at least one sample will cover parts of the code unrelated to those covered by the rest of the suite. Besides, method call frequency is considered a non-structural coverage metric. To mitigate this threat, we also evaluate our test suites using a structural metric, in particular, (frequency of) call sequences.

6.6 Limitations

Context sensitivity: Although, our probabilistic grammar learning approach captures the distribution of input properties, the learned input distribution is limited to production choices at the syntactic level. This approach does not handle context-sensitive dependencies such as the order, sequences or repetitions of specific input elements. However, our approach can be extended to learn contextual dependencies, e.g. by learning sequences of elements using N-grams [275] or hierarchies of elements using k-paths [276].

Input Constraints: Beyond lexical and syntactical validity, structured inputs often contain input semantics such as checksums, hashes, encryption, or references. Context-free grammars, as applied in this work, do not capture such complex input constraints. Automatically learning such input constraints for test generation is a challenging task [277]. In the future, we plan to automatically learn input constraints to drive test generation, e.g. using attribute grammars.

6.7 Related Work

Generating software tests. The aim of *software test generation* is to find a sample of inputs that induce executions that sufficiently cover the possible behaviors of the program—including undesired behavior. Modern software test generation relies, as surveyed by Anand et al. [66] on *symbolic code analysis* to solve the path conditions leading to uncovered code [67, 68, 69, 70, 71, 72, 73, 74],

search-based approaches to systematically evolve a population of inputs towards the desired goal [75, 76, 77, 78], random inputs to programs and functions [79, 80] or a combination of these techniques [81, 82, 83, 84, 85]. Additionally, machine learning techniques can also be applied to create test sequences [86, 87]. All these approaches have in common that they do not require an additional model or annotations to constrain the set of generated inputs; this makes them very versatile, but brings the risk of producing false alarms—failing executions that cannot be obtained through legal inputs.

Grammar-based test generation. The usage of grammars as *producers* was introduced in 1970 by Hanford in his *syntax machine* [88]. Such producers are mainly used for testing compilers and interpreters: CSmith [89] produces syntactically correct C programs, and LANGFUZZ [90] uses a JavaScript grammar to parse, recombine, and mutate existing inputs while maintaining most of the syntactic validity. GENA [91, 92] uses standard symbolic grammars to produce test cases and only applies stochastic annotation during the derivation process to distribute the test cases and to limit recursions and derivation depth. Grammar-based white-box fuzzing [93] combines grammar-based fuzzing with symbolic testing and is now available as a service from Microsoft. As these techniques operate with system inputs, any failure reported is a true failure—there are no false alarms. None of the above approaches use probabilistic grammars, though.

Probabilistic grammars. The foundations of probabilistic grammars date back to the earliest works of Chomsky [94]. The concept has seen several interactions and generalizations with physics and statistics; we recommend the very nice article by Geman and Johnson [95] as an introduction. Probabilistic grammars are frequently used to analyze ambiguous data sequences—in computational linguistics [96] to analyze natural language, and in biochemistry [97] to model and parse macromolecules such as DNA, RNA, or protein sequences. Probabilistic grammars have been used also to model and produce input data for specific domains, such as 3D scenes [98] or processor instructions [99].

The usage of probabilistic grammars for test generation seems rather straightforward, but is still uncommon. The *Geno* test generator for .NET programs by Lämmel and Schulte [100] allowed users to specify probabilities for individual production rules. Swarm testing [101, 102] uses statistics and a variation of random testing to generate tests that deliberately targets or omits features of interest. These approaches, in contrast to the one we present in this chapter, does not use existing samples to learn or estimate probabilities. The approach by Poulding et al. [103, 104] uses a stochastic context-free grammar for statistical testing. The goal of this work is to correctly imitate the operational profile and consequently the generated test cases are similar to what one would expect during normal operation of the system. The test case generation [105, 106] and failure reproduction [107] approaches by Kifetew et al. combine probabilistic grammars with a search-based testing approach. In particular, like our work, StGP [105] also learns stochastic grammars from sample inputs.

Our approach aims to generate inputs that induce (dis)similar program behaviors as the sample inputs. In contrast to our work, StGP [105] is focused on evolving and mutating the learned grammars to *improve code coverage*. Although, StGP’s goal of generating realistic inputs is very similar to our “common inputs” strategy (see **RQ1**), our approach can further generate realistic inputs that are dissimilar to the sample inputs (see **RQ2**). Meanwhile, StGP is not

capable of generating dissimilar inputs.

Mining probabilities. Related to our work are approaches that mine grammar rules and probabilities from existing samples. Patra and Pradel [278] use a given parser to mine probabilities for subsequent fuzz testing and to reduce tree-based inputs for debugging [279]. Their aim, however, is not to produce inputs that would be similar or dissimilar to existing inputs, but rather to produce inputs that have a higher likelihood to be syntactically correct. This aim is also shared by two *mining* approaches: GLADE [280] and Learn&Fuzz [281], which learn producers from large sets of input samples even without a given grammar.

All these approaches, however, share the problem of producing only “common inputs”—they can only focus on common features rather than uncommon features, creating a general “tension between conflicting learning and fuzzing goals” [281]. In contrast, our work can specifically focus on “uncommon inputs”—that is, the complement of what has been learned.

Like us, the Skyfire approach [282] aims at also leveraging “uncommon inputs” for probabilistic fuzzing. Their idea is to learn a probabilistic distribution from a set of samples and use this distribution to generate seeds for a standard fuzzing tool, namely AFL [283]. Here, favoring low probability rules is one of many heuristics applied besides low frequency, low complexity, or production limits. Although the tool has shown good results for XML-like languages, results for other, general grammar formats such as JavaScript are marked as “preliminary” only, though.

Mining grammars. Our approach requires a grammar that can be used both for parsing and producing inputs. While engineering such a grammar may well pay off in terms of better testing, it is still a significant investment in the case of specific domain inputs where such a grammar might not be immediately available. Mining input structures [284], as exemplified using the above GLADE [280] and Learn&Fuzz [281] approaches, may assist in this task. AUTOGRAM [285] and MIMID [286] mine human-readable input grammars exploiting structure and identifiers of a program processing the input, which makes them particularly promising.

6.8 Discussions and Future Work

This chapter empirically show that *learning input distribution from sample inputs in the wild is useful to drive test generation*. We have presented an approach that allows engineers, using a grammar and a set of input samples, to generate instances that are either similar or dissimilar to these samples. Similar samples are useful, for instance, when learning from failure-inducing inputs; while dissimilar samples could be used to leverage the testing approach to explore previously uncovered code. Our approach provides a simple, general, and cost-effective means to generate test cases that can then be targeted to the commonly used portions of the software, or to the rarely used features.

Despite their usefulness for test case generation, grammars (including probabilistic grammars) still have a lot of potential to explore in research, and a lot of ground to cover in practice. In the future, we plan to investigate the following topics:

Deep models. At this point, our approach captures probabilistic distributions only at the level of individual rules. However, probabilistic distributions could also capture the occurrence of elements in particular *contexts*, and differentiate between them. For instance, if a "+"

symbol rarely occurs within parentheses, yet frequently outside of them, this difference would, depending on how the grammar is structured, not be caught by our approach. The domain of computational linguistics [96] has introduced a number of models that take context into account. In our future work, we shall experiment with deeper context models, and determining their effect on capturing common and uncommon input features.

Grammar learning. The big cost of our approach is the necessity of a formal grammar for both parsing and producing—a cost that can boil down to 1–2 programmer days if a formal grammar is already part of the system (say, as an input file for parser generators), but also extend to weeks if it is not. In the future, we will be experimenting with approaches that *mine grammars* from input samples and programs [285, 286] with the goal of extending the resulting grammars with probabilities for probabilistic fuzzing.

Debugging. Mined probabilistic grammars could be used to characterize the features of failure-inducing inputs, separating them from those of passing inputs. Statistical fault localization techniques [1], for instance, could then identify input elements most likely associated with a failure. Generating “common inputs”, as in this work, and testing whether they cause failures, could further strengthen correlations between input patterns and failures, as well as narrow down the circumstances under which the failure occurs.

We provide the source code of our parsers, production tools, the raw input samples, as well as all obtained data and processed charts as a replication package:

<https://tinyurl.com/inputs-from-hell>

Conclusion

In this thesis, we propose an *evidence-driven approach* to address challenges in software testing and debugging. The main idea of this approach is to gather empirical evidence from software practice to guide and support the debugging activities of software developers. We conclude this dissertation with a summary of our work and its contributions. We also discuss future research opportunities and elicit on possible directions for further improvement and investigation.

7.1 Summary

We provide a summary of the challenges addressed in this work with respect to the thesis statement: “*Software testing and debugging should be driven by empirical evidence collected from software practice*”. Specifically, to evaluate this thesis statement, we pose these scientific questions:

- 1. How do developers debug and repair software bugs?** We have conducted an empirical study to collect data on the tools and strategies employed by developers while debugging real faults. Our study includes a survey of over 200 developers and a human study where we observe 12 developers while they debug 27 real bugs. In this study, we collected data on the debugging needs of developers and the reasons for developers (non-)adoption of debugging aids. Indeed, we have found that there is a gap between the needs of developers and the tools provided by researchers.
- 2. What is the most effective automated fault localization (AFL) technique?** We have evaluated the effectiveness of the state-of-the-art fault localization techniques using hundreds of real faults. In particular, we evaluated the performance of 18 most effective statistical debugging formulas against program slicing. We also evaluate the impact of error type (artificial or real faults) and the number of faults (single or multiple faults) on the effectiveness of these AFL methods. In our evaluation, we found that dynamic slicing is best suited for diagnosing single faults, while, statistical debugging performs better on multiple faults.
- 3. How can we automatically debug and repair real-world invalid inputs?** In the context of input debugging, we evaluate the prevalence and causes of invalidity in inputs, using thousands of real-world input files. In our evaluation, we found that four percent of input files in the wild are invalid, they were either rejected by at least one subject program or their

input grammar. We have also identified a number of causes of input invalidity, such as wrong syntax and missing or nonconforming elements. Many inputs were invalid because of single character errors, such as a deleted character, a missing character or an extraneous character.

4. **Can we leverage real-world sample inputs to guide test generation?** We provide an approach to automatically learn the distribution of input elements from sample inputs found in the wild. Our approach employs probabilistic grammars to learn input distribution and applies the learned grammars to generate inputs that are similar or dissimilar to the the initial samples. In our evaluation, our approach effectively generates inputs that reproduce/avoid failures and tests for neighboring program behaviors. During debugging, this is useful for bug reproduction and testing the completeness of bug fixes.

7.2 Contributions

This thesis answers the aforementioned scientific questions and provides a number of tools and methods to aid developers during debugging activities as well as researchers in evaluating debugging and repair tools. Specifically, this dissertation makes the following technical contributions:

DBGBENCH. In our empirical study on debugging in practice, we have collected hundreds of real world fault locations and patches provided by 12 developers while debugging 27 real bugs. We provide the empirical data from this study as a benchmark called **DBGBENCH**. **DBGBENCH** provides details on the debugging needs of developers, as well as the tools and strategies employed by developers when debugging real faults. In addition, it is useful for the automatic evaluation of debuggers and automated repair tools. Our evaluation setup, empirical data and experimental results are available at: <https://dbgbench.github.io>

Hybrid Fault Localization. We have proposed a hybrid approach that synergistically combines the strengths of both dynamic slicing and statistical debugging, our hybrid approach builds on the empirical evidence from our study on the effectiveness of AFL techniques. This hybrid approach combines the contextual information provided by dynamic slicing and the fault correlation analysis performed by statistical debugging to effectively diagnose faults. Our evaluation showed that our hybrid strategy overcomes the weaknesses of both slicing and statistical fault localization. In our evaluation with hundreds of faults, the best fault localization approach is the hybrid strategy, regardless of the number or type of program faults. The empirical data and results obtained in this evaluation can be found here: <https://tinyurl.com/HybridFaultLocalization>

Maximizing Delta Debugging (*ddmax*). Building on the empirical results from our study on the causes of input invalidity, we provide a black-box technique for automatically diagnosing and repairing invalid inputs via several test experiments. Our maximizing delta debugging algorithm (*ddmax*) (1) identifies which parts of the input data prevent processing, and (2) recover as much of the (valuable) input data as possible. Through experiments, *ddmax* maximizes the subset of the input that can still be processed by the program, thus recovering and repairing as much data as possible. The difference between the original failing input

and the “maximized” passing input includes all input fragments that could not be processed. We provide our tool and findings in an artifact package, the artifact is available at: <https://tinyurl.com/debugging-inputs-icse-2020>

Probabilistic Test Generation (Inputs From Hell). Applying probabilistic grammars as input parsers, we show how to learn input distributions from input samples, allowing to create inputs that are (dis)similar to the sample. Among many use cases, this method allows for the generation of *failure-inducing inputs* – learning from inputs that caused failures in the past gives us inputs that share similar features and thus also have a high chance of triggering bugs. This is useful for bug reproduction and testing the completeness of bug fixes. Our evaluation shows that learning from failure-inducing sample inputs effectively reproduces the same failure and also reveal new failures. The experimental setup, evaluation data and results are available at: <https://tinyurl.com/inputs-from-hell>

7.3 Future Work

This dissertation opens the door for a number of exciting future research opportunities. There are still lots of open problems and research challenges to address in the interplay of software practice and automated debugging. Our future work will focus on the following issues:

Multiple Fault Locations. In contrast to the assumptions of typical debugging and repair tools, most real-world bugs can not be explained, localized or patched by finding one fault location [287, 112]. In the context of automated debugging, this assumption is called *perfect bug understanding*. It is the assumption that finding and examining a single faulty statement in the program is sufficient to detect, understand, explain and fix the bug [178]. A few researchers have shown that this assumption does not hold in software practice [112], in fact it has been shown to impede the effectiveness of debugging aids and automated repair techniques [287]. The results from our empirical study on debugging practice (in Chapter 3) also show that the perfect bug understanding assumption does not hold in reality. Indeed, for most bugs, developers provided bug diagnoses and fixes that span more than one contiguous fault location. For instance, we found that most (over 50% of) developer-provided diagnoses span multiple (three to four) contiguous code locations and multiple (10 statements) faulty LoC (, on average). This result has major implications for debuggers and repair tools in practice, since most tools assume *perfect bug understanding*.

To address this problem, we plan to conduct an empirical study to investigate the impact of multi-line faults and multiple (contiguous) fault locations on the productivity of developers and the effectiveness of debugging techniques. It is pertinent to examine the impact of the perfect bug understanding assumption on the effectiveness of debugging aids and its cost in terms of developer productivity and time. Besides, this assumption has serious implications for the (previous) evaluation of debugging and repair tools. We plan to first empirically investigate the impact of this assumption in realistic debugging settings with developers, using real-world bugs with multi-line faults and multiple contiguous fault locations. We

would then employ the empirical evidence collected from this study to build tools that address this challenge in practical debugging and repair settings.

Advanced Input Debugging. We are investigating a number of open problems in input debugging, in particular, *how to leverage program features to improve input repair*. To address this problem, we plan to develop a *white-box input repair* approach. The goal is to leverage program semantics to improve input diagnosis and repair.

First, we plan to empirically investigate the presence of program invariants that correlates with input (in)validity in software practice, using a number of subject programs and thousands of inputs. Applying the lessons learned from this investigation, we would develop a white-box input repair algorithm that automatically learns program invariants (e.g. coverage-based metrics) that correlates with input (in)validity. One can employ such invariants as semantic checks for input diagnosis and repair, for instance, via test oracle checks for the coverage of error-handling method calls. White-box input repair is important to effectively repair invalid inputs for programs that silently handle failures, e.g. without triggering a program crash.

Bibliography

- [1] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.
- [2] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.
- [3] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2017.
- [4] Josep Silva. A survey on algorithmic debugging strategies. *Advances in engineering software*, 42(11):976–991, 2011.
- [5] Richard Torkar and Stefan Mankefors. A survey on testing and reuse. In *Proceedings 2003 Symposium on Security and Privacy*, pages 164–173. IEEE, 2003.
- [6] Mats Grindal, Jeff Offutt, and Sten F Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005.
- [7] Per Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006.
- [8] Robert V Binder. Testing object-oriented software: a survey. *Software Testing, Verification and Reliability*, 6(3-4):125–252, 1996.
- [9] Heiko Koziol. The role of experimentation in software engineering. 2006.
- [10] Walter F Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A Heinz. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28(1):9–18, 1995.
- [11] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating & improving fault localization techniques. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017*, 2017.
- [12] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013*, pages 314–324, 2013.

- [13] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE'15, pages 913–923, 2015.
- [14] Marcel Böhme and Abhik Roychoudhury. Corebench: Studying complexity of regression errors. In *Proceedings of the 23rd ACM/SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, pages 105–115, 2014.
- [15] Qianqian Wang, Chris Parnin, and Alessandro Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 1–11. ACM, 2015.
- [16] Walter F Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998.
- [17] Marvin V Zelkowitz and Dolores R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, 1998.
- [18] Victor R Basili. The role of experimentation in software engineering: past, current, and future. In *Proceedings of IEEE 18th International Conference on Software Engineering*, pages 442–449. IEEE, 1996.
- [19] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, pages 1–28, 2016.
- [20] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. Inputs from hell: Learning input distributions for grammar-based test generation. *Transaction on Software Engineering (TSE)*, 2020.
- [21] Ezekiel O Soremekun. Debugging with probabilistic event structures. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 437–440. IEEE, 2017.
- [22] Marcel Böhme, Ezekiel Olamide Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Wo ist der fehler und wie wird er behoben? ein experiment mit softwareentwicklern. *Software Engineering und Software Management (SE) 2018*, 2018.
- [23] IEEE. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.
- [24] Henry Lieberman. The debugging scandal and what to do about it (introduction to the special section). *Communications of the ACM*, 40(4):26–29, 1997.
- [25] Abhik Roychoudhury. Debugging as a science, that too, when your program is changing. *Electronic Notes in Theoretical Computer Science*, 266:3–15, 2010.
- [26] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, Feb 2013.

-
- [27] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, 2002.
 - [28] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, 2005.
 - [29] Manu Jose and Rupak Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 437–446, 2011.
 - [30] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: An approach to debugging evolving programs. *ACM Transactions on Software Engineering and Methodology*, 21(3):19:1–19:29, July 2012.
 - [31] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, pages 253–267, 1999.
 - [32] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 1–10, 2002.
 - [33] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, January 2012.
 - [34] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 43–54, 2015.
 - [35] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, 2013.
 - [36] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 257–269, 2015.
 - [37] Andrew Ko. The black hole of software engineering research. <https://blogs.uw.edu/ajko/2015/10/05/the-black-hole-of-software-engineering-research/>, Oct. 2015. Accessed: 2017-01-12.
 - [38] L. Briand. Embracing the engineering side of software engineering. *IEEE Software*, 29(4):96–96, July 2012.

- [39] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, August 2016.
- [40] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP’05*, pages 528–550, 2005.
- [41] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners’ expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*, pages 165–176, 2016.
- [42] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.
- [43] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering, ICSE ’81*, pages 439–449, 1981.
- [44] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.
- [45] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Trans. Softw. Eng. Methodol.*, 16(2), April 2007.
- [46] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE ’05*, pages 273–282, 2005.
- [47] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, October 1988.
- [48] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, PLDI ’90*, pages 246–256, 1990.
- [49] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser, and Saul London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance, ICSM ’93*, pages 348–357, 1993.
- [50] Tibor Gyimóthy, Árpád Beszédes, and Istán Forgács. An efficient relevant slicing method for debugging. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-7*, pages 303–321, 1999.
- [51] Dawei Qi, Hoang D. T. Nguyen, and Abhik Roychoudhury. Path exploration based on symbolic output. *ACM Trans. Softw. Eng. Methodol.*, 22(4):32:1–32:41, October 2013.
- [52] Hiralal Agrawal, Richard A DeMillo, and Eugene H Spafford. Efficient debugging with slicing and backtracking. Technical report, Purdue University, 1990.

-
- [53] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, 2005.
 - [54] Alice X Zheng, Michael I Jordan, Ben Liblit, and Alex Aiken. Statistical debugging of sampled programs. In *Advances in Neural Information Processing Systems*, pages 9–11, 2003.
 - [55] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, 2013.
 - [56] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziyang Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 254–265, 2014.
 - [57] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.
 - [58] James Clause and Alessandro Orso. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 249–260. ACM, 2009.
 - [59] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151. ACM, 2006.
 - [60] Chad D. Sterling and Ronald A. Olsson. Automated bug isolation via program chipping. *Software: Practice and Experience*, 37(10):1061–1086, 2007.
 - [61] Martin C. Rinard. Living in the comfort zone. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 611–622, New York, NY, USA, 2007. ACM.
 - [62] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic input rectification. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 80–90, Piscataway, NJ, USA, 2012. IEEE Press.
 - [63] Tomasz Kuchta, Cristian Cadar, Miguel Castro, and Manuel Costa. Doccovery: Toward generic automatic document recovery. In *International Conference on Automated Software Engineering (ASE 2014)*, pages 563–574, 9 2014.
 - [64] Jan Scheffczyk, Uwe M Borghoff, Peter Rödig, and Lothar Schmitz. S-dags: Towards efficient document repair generation. In *Proc. of the 2nd Int. Conf. on Computing, Communications and Control Technologies*, volume 2, pages 308–313, 2004.
 - [65] Paul Eric Ammann and John C Knight. Data diversity: An approach to software fault tolerance. *Ieee transactions on computers*, (4):418–425, 1988.

- [66] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.
- [67] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 97–107, New York, NY, USA, 2004. ACM.
- [68] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2329–2344. ACM, 2017.
- [69] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.
- [70] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, 2008.
- [71] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.
- [72] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 19–32. ACM, 2013.
- [73] Maria Christakis, Peter Müller, and Valentin Wüstholtz. Guiding dynamic symbolic execution toward unverified program executions. In Laura K. Dillon, Willem Visser, and Laurie Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 144–155. ACM, 2016.
- [74] Nikolai Tillmann and Jonathan de Halleux. Pex—white box test generation for .NET. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

-
- [75] Phil McMinn. Search-based software testing: Past, present and future. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 153–163, Washington, DC, USA, 2011. IEEE Computer Society.
- [76] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.
- [77] A. Panichella, F. M. Kifetew, and P. Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, Feb 2018.
- [78] Ke Mao, Mark Harman, and Yue Jia. Sapienz: multi-objective automated testing for android applications. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*, pages 94–105. ACM, 2016.
- [79] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.
- [80] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.
- [81] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.
- [82] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.
- [83] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2329–2344. ACM, 2017.
- [84] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

- [85] Luca Della Toffola, Cristian-Alexandru Staicu, and Michael Pradel. Saying ‘hi!’ is not enough: mining inputs for effective test generation. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 44–49. IEEE Computer Society, 2017.
- [86] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. Automatic text input generation for mobile testing. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017*, pages 643–653. IEEE / ACM, 2017.
- [87] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based GUI testing of Android apps. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017*, pages 245–256. ACM, 2017.
- [88] Kenneth V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.
- [89] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’11*, pages 283–294, New York, NY, USA, 2011. ACM.
- [90] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, Bellevue, WA, 2012. USENIX.
- [91] Hai-Feng Guo and Zongyan Qiu. Automatic grammar-based test generation. In Hüsni Yenigün, Cemal Yilmaz, and Andreas Ulrich, editors, *Testing Software and Systems - 25th IFIP WG 6.1 International Conference, ICTSS 2013, Istanbul, Turkey, November 13-15, 2013, Proceedings*, volume 8254 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2013.
- [92] Hai-Feng Guo and Zongyan Qiu. A dynamic stochastic model for automatic grammar-based test generation. *Softw., Pract. Exper.*, 45(11):1519–1547, 2015.
- [93] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’08*, pages 206–215, New York, NY, USA, 2008. ACM.
- [94] Noam Chomsky. *Syntactic structures*. Mouton, 1957.
- [95] Stuart Geman and Mark Johnson. Probabilistic grammars and their applications. In *International Encyclopedia of the Social & Behavioral Sciences. N.J. Smelser and P.B.*, pages 12075–12082, 2000.

-
- [96] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999.
- [97] Yasubumi Sakakibara, Michael Brown, Richard Hughey, I. Saira Mian, Kimmen Sjölander, Rebecca C. Underwood, and David Haussler. Stochastic context-free grammars for tRNA modeling. *Nucleic Acids Research*, 22(23):5112–5120, 1994.
- [98] Tianqiang Liu, Siddhartha Chaudhuri, Vladimir G. Kim, Qixing Huang, Niloy J. Mitra, and Thomas Funkhouser. Creating consistent scene graphs using a probabilistic grammar. *ACM Trans. Graph.*, 33(6):211:1–211:12, November 2014.
- [99] O. Cekan and Z. Kotasek. A probabilistic context-free grammar based random test program generation. In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 356–359, Aug 2017.
- [100] Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko, editors, *Testing of Communicating Systems*, pages 19–38, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [101] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 78–88, 2012.
- [102] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christy. Generating focused random tests using directed swarm testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 70–81, 2016.
- [103] Robert Feldt and Simon M. Poulding. Finding test data with specific properties via metaheuristic search. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, pages 350–359. IEEE Computer Society, 2013.
- [104] Simon M. Poulding, Robert Alexander, John A. Clark, and Mark J. Hadley. The optimisation of stochastic grammars to enable cost-effective probabilistic structural testing. *Journal of Systems and Software*, 103:296–310, 2015.
- [105] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. Combining stochastic grammars and genetic programming for coverage testing at the system level. In Claire Le Goues and Shin Yoo, editors, *Search-Based Software Engineering - 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings*, volume 8636 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2014.
- [106] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. Generating valid grammar-based test inputs by means of genetic programming and annotated grammars. *Empirical Software Engineering*, 22(2):928–961, 2017.
- [107] Fitsum Meshesha Kifetew, Wei Jin, Roberto Tiella, Alessandro Orso, and Paolo Tonella. Reproducing field failures for programs with complex grammar-based input. In *Seventh*

- IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 163–172. IEEE Computer Society, 2014.
- [108] Marcel Böhme, Ezekiel Olamide Soremekun, Sudipta Chattopadhyay, Emamurho Juliet Ugherughe, and Andreas Zeller. How developers debug software—the dbgbench dataset. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 244–246. IEEE, 2017.
- [109] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 117–128, 2017.
- [110] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014*, pages 437–440, 2014.
- [111] Thomas D. LaToza and Brad A. Myers. Designing useful tools for developers. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU ’11*, pages 45–50, 2011.
- [112] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA*, pages 199–209, 2011.
- [113] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE ’13*, pages 802–811, 2013.
- [114] Wensheng Dou, Shing-Chi Cheung, and Jun Wei. Is spreadsheet ambiguity harmful? detecting and repairing spreadsheet smells due to ambiguous computation. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pages 848–858, 2014.
- [115] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. Automatically generated patches as debugging aids: A human study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, pages 64–74, 2014.
- [116] Website. Dbgbench: From practitioners for researchers. <https://dbgbench.github.io>, 2017. Accessed: 2017-06-30.
- [117] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, August 2002.

-
- [118] Justin Kruger and David Dunning. Unskilled and unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments. *Journal of Personality and Social Psychology*, 77(6):1121–1134, 1999.
- [119] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.
- [120] Sanford Labovitz. Some observations on measurement and statistics. *Social Forces*, pages 151–160, 1967.
- [121] J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.
- [122] Kathy Charmaz. *Constructing grounded theory : a practical guide through qualitative analysis*. Sage Publications, London; Thousand Oaks, Calif., 2006.
- [123] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. How practitioners perceive the relevance of software engineering research. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 415–425, 2015.
- [124] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pages 12–23, 2014.
- [125] Wilhelm Hudson. *Card Sorting*. 2013.
- [126] Paulette Rothbauer. *Triangulation*, pages 892–894. The SAGE Encyclopedia of Qualitative Research Methods, 2008.
- [127] Philip J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education, SIGCSE '13*, pages 579–584, 2013.
- [128] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10*, pages 43–52, 2010.
- [129] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, December 2007.
- [130] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15*, pages 111–124, 2015.
- [131] Alice X. Zheng, Michael I. Jordan, Ben Liblit, and Alexander Aiken. Statistical debugging of sampled programs. In *Proceedings of the Advances in Neural Information Processing Systems 16, NIPS'03*, pages 603–610, 2003.

- [132] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 266–276, 2014.
- [133] Website. Valgrind instrumentation framework. <http://valgrind.org>, 2017. Accessed: 2017-06-30.
- [134] Elaine J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, November 1982.
- [135] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.
- [136] Sebastian Elbaum and David S. Rosenblum. Known unknowns: Testing in the presence of uncertainty. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 833–836, 2014.
- [137] Emanuel Kitzelmann. *Inductive Programming: A Survey of Program Synthesis Techniques*, pages 50–73. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [138] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 24–36, 2015.
- [139] Andrew J. Ko, Thomas D. Latoza, and Margaret M. Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20(1):110–141, February 2015.
- [140] Website. Docker infrastructure – homepage. <http://docker.com/>, 2017. Accessed: 2017-06-30.
- [141] Research Ethics Policy and Advisory Committee, University of Toronto. Guidelines for compensation and reimbursement of research participants, 2017.
- [142] Website. Upwork - hire freelancers and post programming jobs. <https://www.upwork.com/>, 2017. Accessed: 2017-06-30.
- [143] Sanford Labovitz. Some observations on measurement and statistics. *Social Forces*, 46(2):151–160, 1967.
- [144] Jerald Greenberg and Robert Folger. *Experimenter Bias*, pages 121–138. Springer New York, 1988.
- [145] Dewayne E. Perry, Nancy Staudenmayer, and Lawrence G. Votta. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, July 1994.

-
- [146] Pablo Romero, Benedict du Boulay, Richard Cox, Rudi Lutz, and Sallyann Bryant. Debugging strategies and tactics in a multi-representation software environment. *International Journal of Human-Computer Studies*, 65(12):992 – 1009, 2007.
 - [147] Irvin R. Katz and John R. Anderson. Debugging: An analysis of bug-location strategies. *International Journal of Human-Computer Interaction*, 3(4):351–399, December 1987.
 - [148] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.
 - [149] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. Lamelas Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2016.
 - [150] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 306–317, 2014.
 - [151] R. Abreu, P. Zoetewij, and A. J. c. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC’06, pages 39–46, 2006.
 - [152] Shin Hwei Tan and Abhik Roychoudhury. Relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE ’15, pages 471–482, 2015.
 - [153] Adam Kuper and Jessica Kuper. *The Social Science Encyclopedia*. Routledge, 1985.
 - [154] Barbara A. Kitchenham and Shari L. Pfleeger. *Personal Opinion Surveys*, pages 63–92. Springer London, London, 2008.
 - [155] Elizabeth J. Austin, Ian J. Deary, Gavin J. Gibson, Murray J. McGregor, and J. Barry Dent. Individual response spread in self-report scales: personality correlations and consequences. *Personality and Individual Differences*, 24(3):421 – 438, 1998.
 - [156] Xitao Fan, Brent C. Miller, Kyung-Eun Park, Bryan W. Winward, Mathew Christensen, Harold D. Grotevant, and Robert H. Tai. An exploratory study about inaccuracy and invalidity in adolescent self-report surveys. *Field Methods*, 18(3):223–244, 2006.
 - [157] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
 - [158] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.
 - [159] David J Gilmore. Models of debugging. *Acta psychologica*, 78(1):151–172, 1991.
 - [160] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 32(12), 2006.

- [161] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert DeLine, and Gina Venolia. Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 383–392. IEEE, 2013.
- [162] Margaret Ann Francel and Spencer Rugaber. The value of slicing while debugging. *Science of Computer Programming*, 40(2-3):151–169, 2001.
- [163] Andrew J. Ko and Brad A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 151–158, 2004.
- [164] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7(1):49–76, 2002.
- [165] Jonathan Sillito, Kris De Volder, Brian Fisher, and Gail Murphy. Managing software change tasks: An exploratory study. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10–pp. IEEE, 2005.
- [166] Mark Weiser and Jim Lyle. Experiments on slicing-based debugging aids. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 187–197. Ablex Publishing Corp., 1986.
- [167] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, 1981.
- [168] Paul Dan Marinescu and Cristian Cadar. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 235–245, 2013.
- [169] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 334–344, 2013.
- [170] Kiran Lakhotia, Mark Harman, and Phil McMinn. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1098–1105, 2007.
- [171] Ezekiel Soremekun, Lukas Kirschner, Marcel Böhme, and Andreas Zeller. Locating faults with program slicing: An empirical analysis. *Empirical Software Engineering (EMSE)*, 2021.
- [172] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, page preprint, 2016.
- [173] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering*, 12(2):143–160, 2007.

-
- [174] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 33–42, 2005.
- [175] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 831–841, 2017.
- [176] Yanbing Yu, James Jones, and Mary Jean Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 201–210. IEEE, 2008.
- [177] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners’ expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 165–176. ACM, 2016.
- [178] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA ’11, pages 199–209, New York, NY, USA, 2011. ACM.
- [179] Rui Abreu, Peter Zoetewij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, November 2009.
- [180] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 314–324, 2013.
- [181] R. Grissom and J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum, 2005.
- [182] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.
- [183] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Proceedings of 16th International conference on Software engineering*, pages 191–200. IEEE, 1994.
- [184] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.
- [185] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools.

- In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 180–182. IEEE, 2017.
- [186] Alexandre Perez, Rui Abreu, and Marcelo d’Amorim. Prevalence of single-fault fixes and its impact on fault localization. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 12–22. IEEE, 2017.
- [187] Rui Abreu, Peter Zoetewij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE, 2009.
- [188] Alice X Zheng, Michael I Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, pages 1105–1112, 2006.
- [189] Nicholas DiGiuseppe and James A Jones. On the influence of multiple faults on coverage-based fault localization. In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 210–220, 2011.
- [190] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2013.
- [191] W Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. Software fault localization using dstar (d*). In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 21–30. IEEE, 2012.
- [192] Manos Renieris and Steven P Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 30–39. IEEE, 2003.
- [193] Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jinsong Dong. Break the dead end of dynamic slicing: localizing data and control omission bug. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 509–519, 2018.
- [194] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620. IEEE, 2017.
- [195] David Landsberg. *Methods and measures for statistical fault localisation*. PhD thesis, University of Oxford, 2016.
- [196] Lee Naish and Hua Jie Lee. Duals in spectral fault localization. In *2013 22nd Australian Software Engineering Conference*, pages 51–59. IEEE, 2013.
- [197] Sofia Reis, Rui Abreu, and Marcelo d’Amorim. Demystifying the combination of dynamic slicing and spectrum-based fault localization. In *IJCAI*, pages 4760–4766, 2019.

-
- [198] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 114–125. IEEE, 2017.
 - [199] Simon Heiden, Lars Grunske, Timo Kehrer, Fabian Keller, Andre van Hoorn, Antonio Filieri, and David Lo. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Software: Practice and Experience*, 49(8):1197–1224, 2019.
 - [200] Arpit Christi, Matthew Lyle Olson, Mohammad Amin Alipour, and Alex Groce. Reduce before you localize: Delta-debugging and spectrum-based fault localization. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 184–191. IEEE, 2018.
 - [201] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d’Amorim. Fault-localization using dynamic slicing and change impact analysis. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 520–523. IEEE, 2011.
 - [202] Yan Lei, Xiaoguang Mao, Ziyang Dai, and Chengsong Wang. Effective statistical fault localization using program slices. In *2012 IEEE 36th Annual Computer Software and Applications Conference*, pages 1–10. IEEE, 2012.
 - [203] Anbang Guo, Xiaoguang Mao, Deheng Yang, and Shangwen Wang. An empirical study on the effect of dynamic slicing on automated program repair efficiency. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 554–558. IEEE, 2018.
 - [204] Bing Liu, Shiva Nejati, Lionel C Briand, et al. Improving fault localization for simulink models using search-based testing and prediction models. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 359–370. IEEE, 2017.
 - [205] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 261–272, 2017.
 - [206] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 2019.
 - [207] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 177–188, 2016.
 - [208] Ting Shu, Lei Wang, and Jinsong Xia. Fault localization using a failed execution slice. In *2017 International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 37–44. IEEE, 2017.

- [209] Bing Liu, Lucia, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. Simulink fault localization: an iterative statistical debugging approach. *Software Testing, Verification and Reliability*, 26(6):431–459, 2016.
- [210] Franz Wotawa. Fault localization based on dynamic slicing and hitting-set computation. In *2010 10th International Conference on Quality Software*, pages 161–170. IEEE, 2010.
- [211] Birgit Hofer and Franz Wotawa. Spectrum enhanced dynamic slicing for better fault localization. In *ECAI*, volume 12, pages 420–425, 2012.
- [212] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.
- [213] Martin Burger and Andreas Zeller. Minimizing reproduction of software failures. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11*, pages 221–231, 2011.
- [214] Mouna Hammoudi, Brian Burg, Gigon Bae, and Gregg Rothermel. On the use of Delta Debugging to reduce recordings and facilitate debugging of web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 333–344, New York, NY, USA, 2015. ACM.
- [215] Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. Model-based debugging or how to diagnose programs automatically. In *Proceedings of the 15th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: Developments in Applied Artificial Intelligence, IEA/AIE '02*, pages 746–757, London, UK, UK, 2002. Springer-Verlag.
- [216] Manu Jose and Rupak Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, pages 437–446, New York, NY, USA, 2011. ACM.
- [217] Lukas Kirschner, Ezekiel Soremekun, and Andreas Zeller. Debugging inputs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020.
- [218] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. Sapfix: Automated end-to-end repair at scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, pages 269–278. IEEE Press, 2019.
- [219] K Hammond and Victor J. Rayward-Smith. A survey on syntactic error recovery and repair. *Computer Languages*, 9(1):51–67, 1984.
- [220] Microsoft Support. How to recover a lost word document, 2018.
- [221] Blender Foundation. blender.org - home of the blender project - free and open 3d creation software, 2018.

-
- [222] assimp team. The open-asset-importer-lib, 2018.
- [223] The appleseedHQ Organization. appleseed - a modern, open-source production renderer, 2018.
- [224] Stephen Dolan. Command-line json processor, 2018.
- [225] json simple. A simple java toolkit for JSON, 2018.
- [226] Ralf Sternberg. minimal-json - a fast and small json parser and writer for java, 2018.
- [227] AT&T Labs Research. Graphviz - graph visualization software, 2018.
- [228] Gephi. The open graph viz platform, 2018.
- [229] GitHub. Grammars written for ANTLR v4, 2018.
- [230] University of Utah. Wavefront obj specification, 2003.
- [231] Douglas Crockford. Ecma-404 the json data interchange standard, 2017.
- [232] Eleftherios Koutsofios, Stephen North, et al. Drawing graphs with dot. Technical report, Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, 1991.
- [233] GitHub Inc. Rest api v3, 2018.
- [234] ANTLR 4.7.1 API JavaDocs. Class defaulterrorstrategy, 2018.
- [235] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.
- [236] Johannes Buchner. Imagehash 4.0, 2017.
- [237] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.
- [238] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. *SIGPLAN Not.*, 38(11):78–95, October 2003.
- [239] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 176–185, May 2005.
- [240] Ishtiaque Hussain and Christoph Csallner. Dynamic symbolic data structure repair. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 2, pages 215–218. IEEE, 2010.
- [241] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis, ISSTA '06*, pages 233–244, New York, NY, USA, 2006. ACM.

- [242] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: A tool for repairing complex data structures. In *Proceedings of the 30th international conference on Software engineering*, pages 855–858. ACM, 2008.
- [243] Roland C Backhouse. *Syntax of programming languages: theory and practice*. Prentice-Hall, Inc., 1979.
- [244] S. O. Anderson and Roland Carl Backhouse. Locally least-cost error recovery in earley’s algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(3):318–347, 1981.
- [245] Carl Cerecke. Locally least-cost error repair in LR parsers. 2003.
- [246] S. O. Anderson, Roland Carl Backhouse, E. H. Bugge, and CP Stirling. An assessment of locally least-cost error recovery. *The Computer Journal*, 26(1):15–24, 1983.
- [247] Michael Burke and Gerald A. Fisher Jr. *A practical method for syntactic error diagnosis and recovery*, volume 17. ACM, 1982.
- [248] Jon Mauney and Charles N. Fischer. A forward move algorithm for LL and LR parsers. *ACM SIGPLAN Notices*, 17(6):79–87, 1982.
- [249] Tomasz Krawczyk. Error correction by mutational grammars. *Information Processing Letters*, 11(1):9–15, 1980.
- [250] Alfred V Aho and Thomas G Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1(4):305–312, 1972.
- [251] Hui Xiong, Gaurav Pandey, Michael Steinbach, and Vipin Kumar. Enhancing data analysis with noise removal. *IEEE Transactions on Knowledge and Data Engineering*, 18(3):304–319, 2006.
- [252] Mauricio A Hernández and Salvatore J Stolfo. The merge/purge problem for large databases. *ACM Sigmod Record*, 24(2):127–138, 1995.
- [253] Ravali Pochampally, Anish Das Sarma, Xin Luna Dong, Alexandra Meliou, and Divesh Srivastava. Fusing data with correlations. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 433–444, 2014.
- [254] Helena Galhardas, Daniela Florescu, Dennis Shasha, and Eric Simon. Ajax: An extensible data cleaning tool. *ACM Sigmod Record*, 29(2):590, 2000.
- [255] Lukasz Golab, Howard Karloff, Flip Korn, and Divesh Srivastava. Data auditor: Exploring data quality and semantics using pattern tableaux. *Proceedings of the VLDB Endowment*, 3(1-2):1641–1644, 2010.
- [256] Shawn R Jeffery, Gustavo Alonso, Michael J Franklin, Wei Hong, and Jennifer Widom. A pipelined framework for online cleaning of sensor data streams. In *22nd International Conference on Data Engineering (ICDE’06)*, pages 140–140. IEEE, 2006.

-
- [257] Vijayshankar Raman and Joseph M Hellerstein. Potter’s wheel: An interactive data cleaning system. In *VLDB*, volume 1, pages 381–390, 2001.
- [258] Dominik Luebbbers, Udo Grimmer, and Matthias Jarke. Systematic development of data mining-based data quality tools. In *Proceedings 2003 VLDB Conference*, pages 548–559. Elsevier, 2003.
- [259] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. Data debugging with continuous testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 631–634, 2013.
- [260] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. Preventing data errors with continuous testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 373–384, 2015.
- [261] Xiaolan Wang, Mary Feng, Yue Wang, Xin Luna Dong, and Alexandra Meliou. Error diagnosis and data profiling with data x-ray. *Proceedings of the VLDB Endowment*, 8(12):1984–1987, 2015.
- [262] Matthias Hörschele and Andreas Zeller. Mining input grammars with autogram. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 31–34. IEEE Press, 2017.
- [263] Rahul Gopinath, Björn Mathis, Mathias Hörschele, Alexander Kampmann, and Andreas Zeller. Sample-free learning of input grammars for comprehensive software fuzzing. *arXiv preprint arXiv:1810.08289*, 2018.
- [264] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.
- [265] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
- [266] S. Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, Sep 2000.
- [267] Kelly O’Hair. HPROF: a Heap/CPU profiling tool in J2SE 5.0. *Sun Developer Network, Developer Technical Articles & Tips*, 28, 2004.
- [268] Robert Feldt, Richard Torkar, Tony Gorschek, and Wasif Afzal. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *First International Conference on Software Testing Verification and Validation, ICST 2008*, pages 178–186. IEEE Computer Society, 2008.
- [269] Qingkai Shi, Zhenyu Chen, Chunrong Fang, Yang Feng, and Baowen Xu. Measuring the diversity of a test set with distance entropy. *IEEE Trans. Reliability*, 65(1):19–27, 2016.

- [270] Wei Jin and Alessandro Orso. Bugredux: reproducing field failures for in-house debugging. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 474–484. IEEE, 2012.
- [271] Andreas Buja, Dianne Cook, Heike Hofmann, Michael Lawrence, Eun-Kyung Lee, Deborah F Swayne, and Hadley Wickham. Statistical inference for exploratory data analysis and model diagnostics. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 367(1906):4361–4383, 2009.
- [272] Yanqin Fan. Testing the goodness of fit of a parametric density function by kernel method. *Econometric Theory*, 10(2):316–356, 1994.
- [273] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In Lionel C. Briand and Alexander L. Wolf, editors, *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, pages 85–103. IEEE Computer Society, 2007.
- [274] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.
- [275] Marc Damashek. Gauging similarity with n-grams: Language-independent categorization of text. *Science*, 267(5199):843–848, 1995.
- [276] Nikolas Havrikov and Andreas Zeller. Systematically covering input structure. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE '19*, page 189–199. IEEE Press, 2019.
- [277] Michaël Mera. Mining constraints for grammar fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 415–418, 2019.
- [278] Jibesh Patra and Michael Pradel. Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. Technical Report TUD-CS-2016-14664, Technical University of Darmstadt, November 2016.
- [279] Satia Herfert, Jibesh Patra, and Michael Pradel. Automatically reducing tree-structured test inputs. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 861–871. IEEE Computer Society, 2017.
- [280] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*, pages 95–110, New York, NY, USA, 2017. ACM.
- [281] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, pages 50–59, Piscataway, NJ, USA, 2017. IEEE Press.

- [282] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy, SP 2017*, pages 579–594. IEEE Computer Society, 2017.
- [283] Michal Zalewski. American fuzzy lop. <http://lcamtuf.coredump.cx/afl/>, 2018. Accessed: 2018-01-28.
- [284] Zhiqiang Lin and Xiangyu Zhang. Deriving input syntactic structure from execution. In Mary Jean Harrold and Gail C. Murphy, editors, *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 83–93. ACM, 2008.
- [285] Matthias Hörschele and Andreas Zeller. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 720–725, New York, NY, USA, 2016. ACM.
- [286] Rahul Gopinath, Björn Mathis, and Andreas Zeller. Mining input grammars from dynamic control flow. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2020*, 2020.
- [287] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 691–701, 2016.

Appendices

List of Figures

1	Dynamic slicing illustrated [46]: The <code>middle</code> return value in Line 15 is the slicing criterion and Line 8 is the faulty statement.	10
2	Slicing Example: Nodes are statements in each line of the <code>middle</code> program (Figure 1). Control-dependencies are dashed lines while data dependencies are shown as concrete lines.	11
3	Statistical Fault Localization Example: The faulty line 8 and its scores are in bold red	12
4	Countries associated with the IP addresses of our survey respondents	21
5	Boxplots of the time spent reproducing, diagnosing, and fixing bugs. The boxplot at the top shows the reported time spent debugging as a percentage of the development time. The boxplots below show the reported time spent on each subtask as a percentage of debugging time.	23
6	Boxplots of respondent’s familiarity with the code they debug (left) and of the frequency with which they debug other people’s code (right).	24
7	Stacked histograms showing how often respondents use the listed debugging techniques and associated tools. For instance, the first row can be read as follows: About 5% of respondents use trace-based debugging never or rarely, 15% sometimes, 50% often, and 30% always.	25
8	Examples of responses to our Question 16: “If you could design an automated bug diagnosis tool that explains the reproduced bug to you, what would it output?” . . .	26
9	Examples of responses to our optional Question 24: “What do you expect from an automated diagnosis tool?”	27
10	Examples of responses to our Question 15: “If you had to review an (auto-generated) bug fix / software patch, which properties must it have to be acceptable?”	28
11	Examples of responses to our Question 25: “What do you expect from an automated bug fixing tool?”	29
12	Venn Diagram showing the set of respondents which oppose the full automation of either bug diagnosis (no auto-diagnosis) or bug fixing (no auto-repair).	30
13	Responses in favour of debugging automation	31
14	Responses against debugging automation.	33
15	Screenshot of the provided virtual environment.	35
16	Questions on the fault locations and bug diagnosis	37
17	Questions on generating the software patch	38

18	Relationship between average time spent and difficulty perceived for patching and diagnosing a bug. Each point is one of 27 bugs, the shape of which determines its bug type (i.e., crash, functional error, infinite loop, or resource leak).	39
19	Boxplots showing the number of contiguous regions per bug diagnosis (left), the number of statements per diagnosis (middle), and the number of statements per contiguous region (right). For example, file.c:12-16,20 specifies six statements in two contiguous regions.	40
20	Boxplots showing the proportional patch correctness and plausibility. For instance, if the proportional patch correctness for an error is 50%, then half of the patches submitted for this error are correct. The boxplot shows the proportional plausibility and correctness <i>over all 27 errors</i>	42
21	Histogram showing reasons why 108 of patches that were submitted by participants failed our code review.	42
22	Diagnosis strategies for different error types.	43
23	Proportional agreement on Top-3 most suspicious fault locations showing box plots with jitter overlay (each shape is one of 27 errors).	45
24	Distribution over all errors (of a certain type) of the proportion of participants who believe that a certain error may ever be <i>explained intuitively</i> by a machine. Each shape in the jitter plot overlay represents one error.	47
25	Distribution over all errors (of a certain type) of the proportion of participants who believe that a certain error may ever be <i>fixed reliably</i> by a machine. Each shape in the jitter plot overlay represents one error.	47
26	An excerpt of DBGBENCH. For the error <code>find.66c536bb</code> , we show (a) the bug report and test case that a participant receives to reproduce the error, (b) the bug diagnosis that we consolidated from those provided by participants (including fault locations), and (c) examples of ways how participants patched the error correctly or incorrectly.	48
27	Top-12 most suspicious statements in file parser.c. There are 26 statements with the same suspiciousness (0.98), including parser.c:3109 mentioned by our participants.	49
28	Original developer-patch for bug in Figure 26.	49
29	Statistical debugging illustrated [46]: The <code>middle</code> function takes three values and returns that value which is greater than or equals the smallest and less than or equals the biggest value; however, on the input (2, 1, 3), it returns 1 rather than 2. Statistical debugging reports the faulty Line 8 (in bold red) as the most suspicious one, since the correlation of its execution with failure is the strongest.	62
30	Dynamic slicing illustrated [46]: The <code>middle</code> return value in Line 15 can stem from any of the assignments to <code>m</code> , but only those in Lines 3 and 8 are executed in the failing run. Following back the dynamic dependency immediately gets the programmer to Line 8, the faulty one.	63

31	Test suite sensitivity of statistical debugging. Let us consider the <code>middle</code> function with fault in line 8 (in bold red), given a small test suite containing two test cases (3, 3, 5) and (2, 1, 3). Then, statistical debugging reports <i>all</i> executed lines 3, 4, 5, 7, 8, and 15 as the “most” suspicious statements, since they are all strongly correlated to the failure.	66
32	Weakness of dynamic slicing: long dependence chain between fault and symptom. This is a variant of the faulty <code>middle</code> function with an operator fault in Line 5 (in bold red). Following back the dynamic dependency gets the programmer to Line 5 (the fault) after inspecting 2–3 statements—(5,6) or (5,7,8) depending on the failing test case.	67
33	Effectiveness of each statistical debugging formula: (a) results are grouped into bars for each family, and (b) cumulative results for all benchmarks (stacked for all families)	78
34	Effectiveness of the most effective statistical formula in (a) each family and (b) the overall average	78
35	Direct comparison of fault localization effectiveness between statistical debugging (Kulczynski2) and dynamic slicing (on single faults) in each benchmark	80
36	Cumulative frequency of the locations to be examined, for dynamic slicing vs. statistical debugging vs. the hybrid approach (on single faults) in each benchmark	82
37	Hybrid sensitiveness to different values of $N \in \{2, 5, 10, 15, 20\}$ showing (a) the cumulative frequency of locations to be examined for all errors (left), and (b) the effectiveness score for each benchmark using the hybrid approach (right).	84
38	Cumulative frequency of the locations to be examined for the hybrid approach, in comparison to statistical debugging (Kulczynski2) and dynamic slicing, for all (Single) Faults. Inspecting only the top two suspicious code locations, Hybrid-2 and dynamic slicing perform similarly (localizing about 47% of errors each); they outperform statistical debugging (39% of errors localized). However, inspecting only the top five locations, Hybrid-2 clearly outperforms slicing and dynamic slicing by localizing 75% of errors, while slicing performs better than statistical debugging (62% vs. 56% of errors).	86
39	Cumulative frequency of the locations to be examined for (a) single-fault and (b) multiple-fault versions of the <code>SIR</code> and <code>IntroClass</code> , using the hybrid approach, statistical debugging (Kulczynski2 and <code>Tarantula</code>) and dynamic slicing	87
40	Effectiveness of each technique for Single and Multiple Fault(s) in <code>SIR</code> and <code>IntroClass</code> : (a) Scores for each benchmark and (b) Scores for both benchmarks	89
41	Effectiveness of each technique on Single and Multiple Faults compared by benchmark <code>SIR</code> and <code>IntroClass</code>	90
42	Failing JSON input	98
43	Failing input reduced with <code>ddmin</code>	98
44	Failing input repaired with <code>ddmax</code>	98
45	Difference between failing and repaired input	99
46	Maximizing Lexical Delta Debugging algorithm	103
47	Workflow of the <code>ddmax</code> evaluation	105

48	Number of Repaired Files for Each Technique	107
49	Data Recovered and Data Loss for all Inputs	108
50	Failing JSON input with missing colon	110
51	Repaired JSON input by <i>ddmax</i>	110
52	Parse tree of Figure 50	111
53	Data Loss Incurred for Invalid Files	113
54	Mean Runtime per Input File for Each Technique	114
55	Derivation tree representing "1 + (2 * 3)"	124
56	Probabilistic grammar G_p , expanding G	124
57	Inputs generated from G_p in Figure 56	124
58	Grammar $G_{p^{-1}}$ inverted from G_p in Figure 56	125
59	Inputs generated from $G_{p^{-1}}$ from Figure 58	126
60	Workflow for the generation of "common inputs" and "uncommon inputs"	129
61	Frequency analysis of call sequences for json-flattener (length=2)	133
62	Call sequences covered by Sample for "common inputs" (PROB) and "uncommon inputs" (INV)	135
63	Call frequency analysis for json-simple-cliftonlabs	136
64	Call frequency analysis for JSONJava	138
65	Call frequency analysis for json-simple	139
66	Number of exceptions triggered by the test suites	141

List of Tables

1	Details of Subject Programs	70
2	Details of Multiple Faults	72
3	Effectiveness of Dynamic Slicing on Single Faults	76
4	Statistical Debugging Effectiveness on Single Faults. Best scores for each (sub)category are in bold ; higher scores are better. For instance, Kulczynski2 is the best performing (single bug optimal) formula for <i>all programs</i> (0.737), on average.	77
5	Effectiveness of Kulczynski2 on single faults, i.e. the most effective statistical formula	79
6	Statistical Tests for the most effective Statistical Debugging Formulas; <i>Odds ratio ψ</i> (all ratios are statistically significant <i>Mann-Whitney U-test</i> < 0.05 for all tests) . . .	81
7	Statistical Tests for all three Fault Localization Techniques: Odds ratio ψ (Mann-Whitney <i>U-test</i> p-values (<i>U</i>) are in brackets), odds ratio with statistically significant p-values determined by Mann-Whitney (<i>U-test</i>) are in bold	83
8	Effectiveness of the Hybrid approach with $N = 2$	84
9	Real vs. Artificial faults	86
10	Effectiveness of all AFL techniques on Single and Multiple Faults for SIR and IntroClass benchmarks. Single Fault Scores are in italics and bracketed, i.e. (<i>Single</i>), while Multiple Fault Scores are in normal text. For multiple faults, the best scores for each (sub)category are in bold ; higher scores are better. For instance, Tarantula is the best performing (popular) statistical debugging formula for <i>all programs with multiple faults with score</i> 0.8269, on average.	88
11	Subject Programs	100
12	Input Grammar Details	100
13	Details of Mined Input Files. We report the <i>cause of input invalidity</i> by showing the <i>number of files rejected by</i> (A) the <i>input grammar</i> , (B) <i>at least one subject program</i> and (C) <i>all subject programs</i>	101
14	<i>ddmax</i> Effectiveness on All Invalid Inputs	108
15	<i>ddmax</i> Efficiency on All Invalid Inputs for each technique (A) Baseline, (B) ANTLR, (C) Lexical <i>ddmax</i> , (D) Syntactic <i>ddmax</i>	109
16	Diagnostic Quality on Real-World Invalid Inputs for Ⓐ <i>ddmin</i> and Ⓑ <i>ddmax</i> diagnoses, and Ⓒ <i>ddmax</i> repair.	115

17	Depth and size of derivation trees for “common inputs” (PROB) and “uncommon inputs” (INV)	130
18	Subject details	132
19	Call Sequence analysis for “common inputs” (PROB) and “uncommon inputs” (INV) for all subject programs	134
20	Method coverage for “common inputs” (PROB) and “uncommon inputs” (INV) . . .	137
21	Sensitivity to training set variance using three different sets ($S \in \{1, 2, 3\}$) of initial samples containing five inputs each	140
22	Sensitivity to the size of the training set using initial sample size $N \in \{1, 5, 10, 50\}$.	140
23	Exception details	142
24	Exceptions induced by “Common Inputs” (PROB), and “Uncommon Inputs” (INV) .	143
25	Reproduced exceptions by sample inputs (SAMP), “failure-inducing inputs” (PROB), inverse of “failure-inducing inputs” (INV) and random grammar-based generation (RAND)	144
26	Reproducing exceptions by “failure-inducing inputs” (PROB), inverse of “failure-inducing inputs” (INV), and random grammar-based test generation (RAND), showing (A) the <i>number of failure-inducing inputs</i> in crawled files, (B) the <i>exception occurrence rate in crawled files</i> , as well as the <i>exception reproduction rate in</i> (C) PROB , (D) RAND and (E) INV	145